UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

ATUALIZAÇÃO DINÂMICA DE SOFTWARE EM SGBDS COM SUPORTE DO MODELO DE COMPONENTES

DISSERTAÇÃO DE MESTRADO

Cleandro Flores De Gasperi

Santa Maria, RS, Brasil

2011

ATUALIZAÇÃO DINÂMICA DE SOFTWARE EM SGBDS COM SUPORTE DO MODELO DE COMPONENTES

por

Cleandro Flores De Gasperi

Dissertação apresentada ao Programa de Pós-Graduação em Informática da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para a obtenção do grau de

Mestre em Computação

Orientador: Profª. Drª. Marcia Pasin

Santa Maria, RS, Brasil

2011

G249a Gasperi, Cleandro Flores De

Atualização dinâmica de software em SGBDS com suporte do modelo de componentes / por Cleandro Flores De Gasperi. – 2011.
56 f.: il.; 30 cm.

Orientador: Marcia Pasin
Dissertação (Mestrado) - Universidade Federal de Santa Maria, Centro de Tecnologia, Programa de Pós-Graduação em Informática, RS, 2011.

1. Sistemas Gerenciadores de Banco de Dados, Atualização Dinâmica de *Software*, Componentes de *Software*. I. Pasin, Marcia. II. Título.

CDU 004.65

Ficha catalográfica elaborada por Cláudia Terezinha Branco Gallotti - CRB 10/1109
Biblioteca Central da UFSM

Endereço Eletrônico: cleandro.gasperi@gmail.com

Universidade Federal de Santa Maria
Centro de Tecnologia
Programa de Pós-Graduação em Informática

A Comissão Examinadora, abaixo assinada,
aprova a Dissertação de Mestrado

ATUALIZAÇÃO DINÂMICA DE SOFTWARE EM SGBDS COM SUPORTE DO MODELO DE COMPONENTES

elaborada por
Cleandro Flores De Gasperi

como requisito parcial para obtenção do grau de
Mestre em Computação

COMISSÃO EXAMINADORA:

Marcia Pasin, Drª.
(Presidente/Orientador)

Deise de Brum Saccol, Drª. (UFSM)

Leandro Krug Wives, Dr. (UFRGS)

Santa Maria, 11 de Outubro de 2011.

AGRADECIMENTOS

Primeiramente a Deus pela benção de chegar ao fim deste trabalho.

A orientação segura e cheia de incentivo da Professora e Amiga Marcia Pasin, que foi fundamental para a elaboração desta dissertação.

À UFSM e ao Centro de Processamento de Dados (CPD), na figura do Diretor Fernando Bordin da Rocha, que proporcionaram a oportunidade de realizar este mestrado, 15 anos após a conclusão da Graduação.

Ao colega Douglas Pasqualin, pela força durante o processo de digitação e formatação desta dissertação com o LaTeX.

Aos meus filhos Arthur e Ellen, que precisaram ficar um pouco mais de tempo sem a presença do pai, mas que sempre me receberam alegres e carinhosos.

À minha esposa e companheira Kelli, que sempre me incentivou e ajudou nos momentos mais difíceis, impedindo que eu desistisse no meio do caminho.

Enfim, a todos aqueles que de alguma forma contribuiram, torceram ou apoiaram nesta caminhada.

Todas as vitórias da criatura são frutos substanciosos da perseverança.
— EMMANUEL —

A nossa maior glória não reside no fato de nunca cairmos,
mas sim em levantarmo-nos sempre depois de cada queda.
— CONFÚCIO —

RESUMO

Dissertação de Mestrado
Programa de Pós-Graduação em Informática
Universidade Federal de Santa Maria

ATUALIZAÇÃO DINÂMICA DE SOFTWARE EM SGBDS COM SUPORTE DO MODELO DE COMPONENTES
AUTOR: CLEANDRO FLORES DE GASPERI
ORIENTADOR: MARCIA PASIN
Local da Defesa e Data: Santa Maria, 11 de Outubro de 2011.

O uso cotidiano da Internet nas mais diversas atividades humanas acaba por criar nos usuários a expectativa de serviços disponíveis a qualquer momento. Muitos destes serviços tem os Sistemas Gerenciadores de Banco de Dados (SGBDs) como ferramenta básica e essencial. Além disso, esses *softwares* estão sujeitos a erros e envelhecimento. Um *software* livre de erros ou que não precise de inovações é uma utopia. Assim, é necessário que o *software* sofra atualizações. Atualmente, os mecanismos para atualização de *software* utilizam *hardware* adicional, uma solução mais cara e complexa, ou optam pela indisponibilição do serviço para os clientes (parada do sistema), que é solução trivial mas ainda eficiente. O que este trabalho traz é a aplicação de técnicas de Atualização Dinâmica de *Software* (ADS) como uma alternativa para atualizar um SGBD sem o uso de *hardware* adicional e a indisponibilização do sistema. Para tanto, propõe-se o desenvolvimento de um SGBD em uma arquitetura hipotética com o suporte de componentes de *software*. Criou-se um protótipo de acordo com a solução proposta, utilizando o modelo de componentes FRACTAL. A avaliação experimental confirmou a viabilidade funcional da solução e que a sobrecarga da implementação em um ambiente controlado foi de aproximadamente 30%. Esta sobrecarga é aceitável, uma vez que se obtem a atualização do SGBD sem a parada total do mesmo.

Palavras-chave: Sistemas Gerenciadores de Banco de Dados, Atualização Dinâmica de *Software*, Componentes de *Software*.

ABSTRACT

Master's Dissertation
Post-Graduate Program in Informatics
Universidade Federal de Santa Maria

DYNAMIC SOFTWARE UPDATE IN DATABASE MANAGEMENT SYSTEMS WITH SUPPORT OF SOFTWARE COMPONENT MODEL

AUTHOR: CLEANDRO FLORES DE GASPERI
ADVISOR: MARCIA PASIN
Defense Place and Date: Santa Maria, October 11th, 2011.

The daily use of Internet services in the most diverse human activities creates in users the expectation of high availability of these services. Many of them have database systems as essential building block. Moreover, those services are subject outcomes such as errors and aging. An error-free software or a non-aging software which does not need innovations is an utopia. Thus, software updating is a required task. Currently, software-updating mechanisms are based on two different solutions: (i) using of additional hardware, an expensive and complex solution, or (ii) service interruption, which is trivial but inefficient. In this work, we explore the application of Dynamic Software Update (DSU) techniques as an alternative to update a Data Base Management System (DBMS) without requiring any additional hardware or service unavailability. Our solution was developed in a hypothetical DBMS architecture with the support of a software component model. A prototype was developed in accordance with this model using FRACTAL. Experimental evaluation confirmed the functional viability of this approach. The implementation overhead in a controlled environment was about 30%, which is acceptable.

Keywords: Database Management System, Dynamic Software Updating, Software Components.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

ACID	Atomicidade, Consistência, Isolamento e Durabilidade
ADL	*Architecture Description Language* - Linguagem de Descrição de Arquitetura
ADS	Atualização Dinâmica de *Software*
API	*Application Programming Interface* - Interface de Programação de Aplicações
AS	Atualizador do Sistema
DBA	*Database Administrator* - Administrador de Banco de Dados
DBC	Desenvolvimento Baseado em Componentes
DSU	*Dynamic Software Update* - Alteração Dinâmica de *Software*
GA	Gerenciador de Atualizações
GI	Gerenciador de Integridade
GMA	Gerenciador do Momento de Atualização
GR	Gerenciador de Requisições
GV	Gerenciador de Versão
IDE	*Integrated Development Enviroment* - Ambiente Integrado de Desenvolvimento
OO	Orientação a Objetos
OSE	*Online Software Evolution* - Evolução de Software *Online*
RSE	*Runtime Software Evolution* - Evolução de *Software* em Tempo de Execução
SGBD	Sistema Gerenciador de Banco de Dados
SO	Sistema Operacional
TI	Tecnologia da Informação
UML	*Unified Modeling Language* - Linguagem Unificada de Modelagem

SUMÁRIO

1 INTRODUÇÃO

1.1 Contexto

O cenário atual da computação está cada vez mais complexo e permeado de conceitos como sistemas computacionais autogerenciáveis, *cloud computing* (computação nas nuvens), redes móveis e computação ubíqua. Em comum, essas novas tecnologias são baseadas em computação distribuída, fazendo uso de rede e em especial a Internet como sua plataforma básica.

Associado a isto, a sociedade contemporânea está sofrendo alterações culturais significativas. O uso cotidiano do computador e da Internet tem modificado substancialmente certas atividades, como o relacionamento com os bancos (pagar contas, sacar dinheiro, etc) e novos hábitos de consumo com o uso de *e-commerce*. Neste sentido, a expectativa dos usuários é de que estes serviços estejam disponíveis sempre, ou seja, 24 horas por dia durante os 7 dias da semana (24 X 7). Eles não devem falhar ou estar indisponíveis exatamente naquele instante em que o usuário deseja utilizá-lo. E mais, o homem contemporâneo que preza pela mobilidade e rapidez (MOREIRA; PONS, 2003) espera por serviços com desempenho satisfatório.

Para muitos serviços, os Sistemas Gerenciadores de Banco de Dados (SGBD) são ferramentas básicas e essenciais. E como qualquer outro tipo de *software* necessita evoluções. De acordo com Mullins (2003) uma tarefa bastante difícil para um DBA (Administrador de Banco de Dados) é manter o SGBD completamente atualizado, ou seja, na sua última versão. As mudanças são necessárias e rápidas. O ciclo de vida para que novos *releases* (com grandes correções e alterações) do SGBD sejam entregues pelos fabricantes é de 18 a 24 meses. Ainda segundo o mesmo autor, a atualização da versão oferece vantagens e riscos. As vantagens são fáceis de serem identificadas, tais como a possibilidade de utilização de novos recursos e funcionalidades, além da correção de *bugs*. Já os riscos podem ser grandes e de difícil identificação. Isto porque as mudanças internas na estrutura do SGBD feitas pelos fabricantes, algumas vezes, podem ser totalmente incompatíveis com a estrutura atual do banco de dados. Aplicações que exijam alta disponibilidade não permitem que o SGBD pare para que seja atualizado. Atualmente, independente do fabricante do SGBD, janelas (períodos de tempo) para manutenção e atualização são necessárias. Estabelecer este procedimento não é uma tarefa trivial.

1.2 Desafio

No cenário descrito anteriormente surgem muitas questões. Por exemplo, qual é a expectativa do usuário quanto à disponibilidade dos mais diversos serviços que lhe são ofertados? E para a equipe responsável por manter as estruturas de apoio a estes serviços funcionando, quando pode parar o sistema para fazer uma manutenção preventiva? Quando pode atualizar as versões dos *softwares* básicos e aplicativos? Que alternativas podem ser viáveis com o objetivo de não parar o sistema, mas ao mesmo tempo continuar o trabalho de manutenção e atualização do mesmo?

As implementações existentes para o problema de atualização da versão do SGBD são baseadas em *hardware* adicional, tornando-se mais caras e complexas. Além do mais, estas soluções são atreladas a aspectos de implementação, o que dificulta a reutilização de uma solução desenvolvida por terceiros. O que se propõe, neste trabalho, é uma alternativa para a atualização da versão do SGBD sem a necessidade de utilização de *hardware* adicional.

O desafio consiste em encontrar uma forma mais simples, de menor custo e com a redução da interferência humana no processo de atualização. Deseja-se mostrar caminhos para que a indústria desenvolva um SGBD capaz de atender um segmento de usuários que não dispõe da estrutura necessária para a implementação de soluções baseadas em *hardware* adicional.

1.3 Contribuição Científica e Objetivos

Atualização Dinâmica de Software (ADS) é uma técnica para permitir que programas em execução possam ser atualizados (novo código e novos dados) sem a necessidade de interromper a sua execução (STOYLE et al., 2007). Sua utilização não é trivial e consiste em um grande desafio, ainda mais quando aplicada a sistemas que necessitam alta disponibilidade e têm alto índice de utilização.

A contribuição deste trabalho de pesquisa é apresentar uma técnica de ADS capaz de resolver o problema de atualização do SGBD sem a necessidade de indisponibilizar o serviço e sem a utilização de *hardware* adicional. Atualmente há carência de soluções genéricas. As grandes organizações, que possuem recursos para tanto, adotam implementações complexas que requerem planejamento, equipe e *hardware* adicional (HICKS; NETTLES, 2005). Já as instituições menores acabam optando por resolver a questão de atualização do SGBD com a indisponibilização do serviço.

Apesar de ADS não ser uma novidade, seu uso na prática é limitado. No contexto atual,

existem soluções que permeiam diferentes áreas da computação como sistemas operacionais (por exemplo, atualização automática do *Windows*, *MacOS*, etc), linguagens de programação e engenharia de *software* (destacando-se aqui o modelo de componentes de *software*). Na literatura não foram encontrados trabalhos que aplicassem estas técnicas a um SGBD.

Acredita-se que a ADS possa se beneficiar do modelo de componentes de *software*. Soluções baseadas em componentes de *software* podem ser genéricas e reaproveitadas.

Assim, objetiva-se, mais precisamente:

1. Obter uma metodologia para atualização automática de um SGBD de forma não atrelada à implementação, sem a necessidade de *hardware* adicional e sem a interrupção total do serviço, com o suporte do modelo de componentes de *software*. Para tanto, é necessário:

   (a) Descrever os módulos do SGBD com o modelo de componentes.

   (b) Obter uma visão arquitetural destes componentes.

2. Descrever um algoritmo para a substituição de componentes, capaz de prever o controle de versões e de identificar o momento ideal para a atualização.

3. Validar experimentalmente a solução com a construção de um protótipo.

1.4 Organização do Texto

Este texto está organizado como segue. No capítulo 2 é feita a revisão da literatura, ilustrando conceitos importantes ao entendimento deste, bem como o estado da arte sobre ADS. O capítulo 3 apresenta a proposta de ADS para SGBD. No capítulo 4 é descrito o protótipo construído para validação e experimentação da solução. A avaliação da solução e do protótipo é feita no capítulo 5. O capítulo 6 encerra este trabalho formulando conclusões e apresentando propostas para trabalhos futuros.

2 REVISÃO DA LITERATURA

Este trabalho de pesquisa propõe uma solução para atualização de um SGBD sem a necessidade de interromper o serviço. Para tanto, alguns elementos precisam ser devidamente apresentados e compreendidos. Assim, neste capítulo serão discutidos:

- Atualização de *Software*: o que é e quais os problemas que podem acontecer durante ou após o processo.
- Atualização Dinâmica de *Software* (ADS): é feita uma revisão de técnicas de ADS e ao final da seção é apresentado um quadro comparativo entre elas.
- Componentes e Arquitetura de *Software*: apresenta-se conceitos de componentes e arquitetura de *software* que são a base da solução apresentada.
- Sistemas Gerenciadores de Banco de Dados (SGBD): esta categoria de *software* é definida e a sua estrutura básica é apresentada. É o ponto focal desta pesquisa.

O leitor familiarizado com estes assuntos pode ir diretamente para o capítulo 3 deste trabalho.

2.1 Atualização de *Software*

A tarefa de substituir o *software* atual por um mais recente recebe o nome de **atualização de *software*** (HICKS; NETTLES, 2005). Esta tarefa é necessária, porque apesar dos esforços realizados pelos desenvolvedores de *software*, entregar ao usuário final um aplicativo livre de problemas e que atenda a todas as necessidades é uma utopia. Uma vez entregue e colocado em produção um determinado *software*, torna-se necessário que novos *releases* e versões sejam disponibilizados a fim de corrigir os eventuais problemas e de implementar as necessidades não cobertas pela implementação anterior. Conforme Rajlich (2000), *release* se refere às pequenas mudanças e correções, enquanto versão se refere às mudanças estratégicas existentes na evolução do *software*.

A experiência mostra que o processo de atualização do *software* pode sofrer problemas como:

- Atualização incompleta: significa que a atualização não foi feita na sua totalidade, fazendo com que o sistema fique inconsistente, gerando erros.

- Inserção de *bugs*: o fato de atualizar a versão de um *software* não garante que este novo *software* esteja livre de *bug*.

Formas de mitigar estes problemas são tratados pela Engenharia de *Software* dentro do processo de construção, manutenção e evolução do *software*, através de duas gerências, a saber: a **Gerência de Configuração**, que é a arte de identificar, organizar e controlar as alterações do *software*; e a **Gerência de Qualidade**, responsável, em síntese, pelos testes do *software* (PRESSMAN, 2001).

A atualização de *software* pode acontecer de duas formas. A primeira consiste da parada total do aplicativo, isto é, ele é retirado dos processos em execução. A seguir o código é substituído pelo mais novo e na sequência liberado para voltar a ser executado. Outro forma é a atualização dinâmica de *software*, que pretende que o processo não seja parado para que a atualização seja efetuada.

2.2 Atualização Dinâmica de Software (ADS)

Embora ADS não seja novidade pois trabalhos mais antigos datam da década de 1970 (FABRY, 1976), soluções genéricas e mais abrangentes são raras.

Em Segal (1993) são destacadas as seguintes propriedades para a ADS:

- Preservar a correção do sistema: significa que o sistema deve se manter íntegro e correto durante e após a atualização. Adicionalmente, é fundamental garantir a transparência da atualização, ou seja, que o usuário nem perceba que ocorreu uma atualização do sistema.

- Minimizar a intervenção humana: esta medida é importante a fim de preservar a correção do sistema. Uma atualização pode exigir uma série de procedimentos em uma ordem específica. Processos automatizados conseguem realizar seus procedimentos com melhor eficiência, minimizando os erros.

- Suportar alterações de baixo nível: refere-se a mudanças mais significativas do sistema, do que simplesmente a substituição de um módulo por um mais novo. Trata do suporte a mudanças de protocolos, interfaces e convenções.

- Suportar reestruturação do código: similar ao anterior, faz referência ao suporte de reestruturações profundas no sistema, como a inclusão e remoção de módulos.

- Suportar programas distribuídos: a atualização de sistemas distribuídos deve considerar além dos aspectos inerentes à atualização em si, aspectos referentes a sua condição distribuída. O processo de atualização deve ser robusto o suficiente para fazer frente aos problemas pertinentes a sistemas distribuídos de larga escala.

- Descartar a necessidade de *hardware* adicional: ao exigir *hardware* adicional para uma atualização, estará se incrementando os custos e diminuindo a portabilidade do sistema.

- Não restringir linguagem e ambiente: deve-se flexibilizar ao máximo para programadores e usuários a escolha dos ambientes operacionais e linguagens utilizadas.

De acordo com Wahler et al. (2009), existem dois tipos de soluções para ADS sem a utilização de *hardware* adicional: soluções baseadas em rotinas de *software* e soluções baseadas em componentes e/ou arquiteturas de *software*. Como a proposta deste trabalho é embasada em uma arquitetura com componentes de *software*, este assunto será detalhado na seção 2.3. A seguir são apresentadas propostas de ADS, conforme esta classificação.

PODUS (*Procedure-Oriented Dynamic Updating System*) (SEGAL; FRIEDER, 1993) é uma ferramenta capaz de promover a ADS que trabalha no nível do Sistema Operacional (SO). É baseada em rotinas de *software*. Para PODUS, um procedimento somente poderá ser atualizado quando estiver inativo. A atualização inicia carregando em outro espaço de memória as novas versões dos procedimentos a serem atualizados. A seguir o programa é interrompido e a sua pilha de execução é analisada. Então através de uma tabela de ligação são feitos os redirecionamentos necessários. Isto significa que, na próxima vez que o procedimento for chamado, será direcionado para a nova versão. Quando todos os procedimentos previstos na atualização estiverem na nova versão, considera-se o programa atualizado.

Na mesma linha de rotinas de *software* está a proposta de Lyu et al. (2001) (*Procedure-Based Dynamic Software Update*) que propuseram a modificação do procedimento de forma direta, sem a utilização de uma tabela de indireções. A nova versão do procedimento a ser atualizado é carregada em uma nova área de memória. Igualmente, o programa é interrompido e sua pilha de execução analisada. Permitida a atualização, então ao invés de fazer o redirecionamento do procedimento com uma tabela de indireção, o endereçamento para a nova versão é feito inserindo no início do procedimento antigo, um desvio para o novo procedimento. Isto significa que, dentro do código da versão antiga do procedimento estará o endereçamento para a nova versão. ADS é feita utilizando chamadas diretas do SO, sem necessidade de *software*

adicional.

Stoyle et al. (2007) apresentam PROTEUS, um núcleo para ADS em linguagens *C-like* que é flexível, seguro e previsível. A atualização pode ser para tipos de dados ou funções. A nova versão é escrita com PROTEUS, onde são explicitados os tipos ou funções a serem substituídas através de sintaxe específica. As alterações são realizadas em tempo de execução. É um modelo baseado em rotinas de *software*.

Uma solução em outro nível de funcionamento é apresentada por Oreizy et al. (1998) (*Architecture-Based Runtime Software Evolution*), cuja proposta é de uma arquitetura capaz de suportar ADS. O principal diferencial está na possibilidade de reutilização da arquitetura. Isto significa que de soluções *ad hoc* passa a existir uma solução geral, que pode ser utilizada por qualquer *software* que for construído de acordo com ela. Esta proposta está baseada no conceito de componentes de *software*. Mas de acordo com a arquitetura há políticas que devem ser respeitadas durante as operações com os componentes, como a necessidade de manutenção da *interface*. É com esta solução que a proposta feita por este trabalho de pesquisa mais se identifica, adotando algumas de suas regras e restrições. Mas também há diferenças, como a de não permitir a reestruturação da arquitetura.

Ainda baseada em componentes de *software* há a proposta de Wang et al. (2006) (*Component-Based Online Software Evolution*), que é de um servidor de aplicações cujos componentes podem ser substituídos dinamicamente. Esta solução não permite remover subcomponentes ou operações de componentes. Isto é necessário para atender a premissa de compatibilidade com a versão que está sendo substituída. Analisando o caso específico de alterações de componentes, o processo de ADS prevê a atualização tanto da interface quanto da implementação. Para tanto é necessário que a linguagem de programação suporte ADS. Diferentes linguagens apresentam diferentes mecanismos. Java é uma linguagem adequada para a implementação de ADS. No caso de Java, existem dois mecanismos importantes: carga da classe (a partir do arquivo *.class*) e reflexão computacional, que é uma técnica de programação que possibilita a capacidade de um sistema atuar sobre sua própria computação e adaptar-se para condições de mudança.

Na tabela 2.1 é apresentado um quadro comparativo das diversas propostas, mostrando o nome da proposta, ano de apresentação, o tipo (de acordo com Wahler et al. (2009)) e o seu nível de atuação.

Percebe-se claramente que as soluções baseadas em componentes de *software* são as mais genéricas, com maior possibilidade de reaproveitamento e flexibilidade. Já as soluções no nível

Proposta	Autores	Ano	Tipo		Nível
			Rotina	Componente	
PODUS	Segal e Frieder	1993	X		SO
Architecture-Based RSE	Oreizy et al.	1998		X	Arquitetura
Procedure-Based DSU	Lyu et al.	2001	X		SO
Component-Based OSE	Wang et al.	2006		X	Serv. Aplicação
PROTEUS	Stoyle et al.	2007	X		Ling. Progr.

Tabela 2.1: Comparativo das propostas de ADS

de SO estão atreladas a detalhes de implementação. O que se espera na solução apresentada neste trabalho é a flexibilidade da proposta de Oreizy et al. (1998) e a segurança e isolamento da proposta de Wang et al. (2006), que permitirá que as atualizações sejam realizadas de forma ACID.

2.3 Componentes e Arquiteturas de *Software*

Segundo Kruchten (1998 apud RESENDE et al., 2007) um componente de *software* representa um elemento significativo e independente, substituível e que cumpre uma função clara e bem definida em uma arquitetura. Além disso, é crucial que este respeite uma interface, que é a sua forma de comunicação com os demais componentes.

Os componentes visam principalmente facilitar o desenvolvimento de aplicativos. Resende (2007) apresenta diversos fatores e vantagens para o uso de componentes de *software* no desenvolvimento de aplicativos, a saber:

- Aumento da qualidade e velocidade no desenvolvimento: *softwares* extensos e complexos podem ser desenvolvidos por organizações diferentes.
- Diversidade de componentes comerciais no mercado, tanto de uso geral quanto de uso específico.
- Aumento do grau de interoperabilidade entre os componentes comerciais.
- Investimentos em pesquisa para o Desenvolvimento Baseado em Componentes (DBC).
- Flexibilidade de desenvolvimento, devendo estar atento mais à especificação do componente e suas interfaces do que na sua implementação inicial. Isto porque uma interface bem definida permite a substituição do componente de forma transparente.
- Reutilização de funcionalidades, técnicas e regras de negócio.

Para Bass et al. (2003), arquitetura de *software* é a estrutura composta por elementos de *software*, as propriedades externas visíveis desses elementos e os relacionamentos entre eles. Evoluindo, tem-se o conceito de arquitetura baseada em componentes de *software* (HEINEMAN; COUNCILL, 2001), que consiste em uma solução eficiente para modelar sistemas computacionais e oferecer uma visão geral dos componentes de um sistema enquanto detalhes de implementação são ocultados. Sob esta ótica, sistemas e programas são vistos como um conjunto de componentes, ou unidades binárias, conectados por interfaces bem definidas.

Neste trabalho, será tomado proveito destas características para implementar ADS em um SGBD.

2.4 Modelo de Componentes FRACTAL

FRACTAL (OW2, 2004) é um modelo de componentes modular e extensível, que pode ser utilizado com várias linguagens de programação para projetar, implementar, executar e reconfigurar sistemas e aplicações. Possui suporte às linguagens *Java* e *C*, e de forma experimental para *.NET*, *SmallTalk* e *Python*.

Um componente FRACTAL é um elemento de execução que está encapsulado, com identificação única e que suporta uma ou mais *interfaces*. A *interface* é um ponto de acesso para um componente e que implementa uma *interface* da linguagem, que representam as operações suportadas (Bruneton *et al.* (2006)). As *interfaces* podem ser de dois tipos, a saber:

- *Cliente* ou *Requerida*: é aquela que invoca o serviço de um outro componente.
- *Servidora* ou *Provida*: é aquela que recebe a solicitação de serviço de um outro componente.

Genericamente, um componente FRACTAL pode ser visualizado com duas camadas, a interna e a externa. A camada externa ou membrana possui as *interfaces* de controle que permitirão a introspecção e a reconfiguração dos componentes. Já a camada interna ou de conteúdo, consiste em um conjunto finito de outros componentes, os **subcomponentes**. As *interfaces* da membrana podem ser internas (acessíveis apenas pelos *subcomponentes*) ou externas (aquelas acessíveis por outros componentes). Um exemplo é apresentado na Figura 2.1.

Para compreender o funcionamento de FRACTAL é preciso assimilar os seguintes conceitos referentes a estrutura dos componentes:

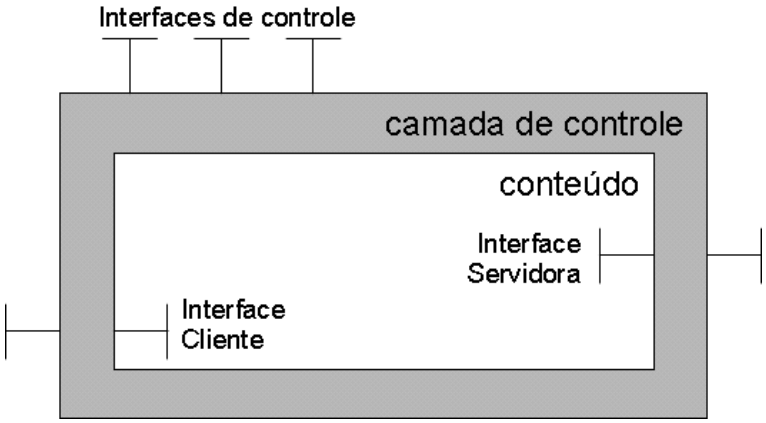

Figura 2.1: Estrutura básica de um componente FRACTAL (OW2, 2004)

- **Componente Primitivo:** é definido pela especificação das interfaces providas (serviços ofertados), as interfaces requeridas e a classe que implementa o componente.
- **Componente Composto:** é definido pela especificação das interfaces providas, as interfaces requeridas, os subcomponentes contidos e as ligações existentes entre eles.
- **Ligações:** são os elementos que explicitam a dependência entre os componentes. É feita no nível das interfaces. Define o caminho de comunicação dos componentes.
- **Controles:** um componente tem controles que podem ser acessados interna ou externamente. São quatro os tipos de controles existentes, a saber:
  - Ciclo de Vida: o controle do ciclo de vida garante a reconfiguração dinâmica dos componentes, pois trata explicitamente da disponiblidade dos mesmos. São duas as operações disponíveis: *start* e *stop*. Basicamente, *start* ativa o componente e *stop* para o componente.
  - Ligações: o controle das ligações é responsável por gerir as dependências entre os componentes. Para realizar a remoção de um componente do sistema, necessariamente, o controle de ligações deve ser utilizado a fim de promover a desconexão entre os componentes, e a futura religação entre eles.
  - Conteúdo: para permitir a adição e remoção de subcomponentes é necessária a implementação do controle de conteúdo. As operações disponíveis neste controle são *addSubComponent* e *removeSubComponent*.
  - Atributo: para configurar propriedades nos componentes se faz necessário o controle de atributos. Normalmente os atributos são tipos primitivos utilizados para

configurar o estado do componente.

Há a possibilidade de descrever a arquitetura com FRACTAL ADL (*Architecture Description Language*) ou desenvolver as aplicações com a utilização da FRACTAL API (*Application Programming Interface*). Estas serão detalhadas na sequência (seções 3.4.1 e 3.4.2), bem como demonstrada a equivalência das suas definições.

2.5 Sistemas Gerenciadores de Banco de Dados

De acordo com Elmasri e Navathe (2005), um Sistema Gerenciador de Banco de Dados (SGBD) é uma coleção de programas de propósito geral que permite aos usuários criar e manter um banco de dados. Ainda conforme estes autores, banco de dados é uma coleção de dados relacionados, onde dados são fatos que podem ser gravados e que possuem um significado.

2.5.1 Estrutura de um SGBD

Classicamente, um SGBD é construído pelos módulos descritos por (RAMAKRISHNAN; GEHRKE, 2003) e sumarizados na Figura 2.2:

- *Analisador Léxico, Sintático e Validador (Parser)*: responsável pela análise léxica e sintática das instruções recebidas. Faz a primeira validação do comando que será processado.
- Otimizador e Executor do Plano de Acesso: recebem do *Parser* a instrução válida. Com as informações que possuem a respeito de como os dados estão armazenados, os índices disponíveis entre outras, determinam um plano de acesso otimizado para resolver a instrução.
- Gerador de Código da Consulta: partindo do plano de execução gerado anteriormente, este módulo produz o código necessário para realização da instrução.
- Mecanismo de Execução: módulo central da arquitetura, tem a função de coordenar a execução das operações necessárias para atendimento do comando. Isto porque é de sua responsabilidade acionar os demais módulos para o processamento da instrução.
- Gerenciador de Transações: acionado pelo Mecanismo de Execução, deve garantir a integridade da transação, controlando o seu início e fim (confirmada (*commit*) ou cancelada (*rollback*)). Uma transação é um programa em execução que forma uma unidade lógica de

trabalho (ELMASRI; NAVATHE, 2005). Deve ser atômica, consistente, isolada e durável (ACID).

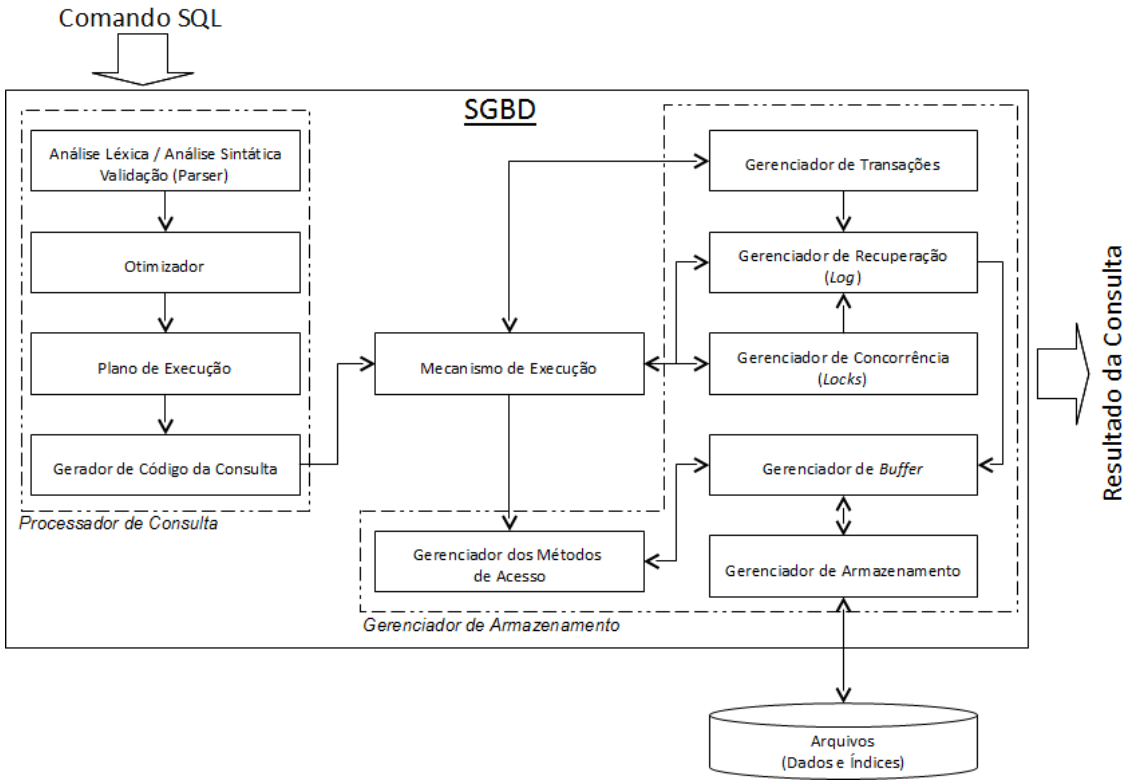

Figura 2.2: Arquitetura básica de um SGBD (adaptado de Silberschatz et al. (2006) e Ramakrishnan e Gehrke (2003))

- Gerenciador de Recuperação: acionado a partir do Mecanismo de Execução, também recebe informações do Gerenciador de Transações. Sua função é controlar operações e valores que serão armazenados no *log* da transação, bem como recuperar esta informação, se necessário, em caso da transação falhar ou ser cancelada (*rollback*).

- Gerenciador de Concorrência: uma função importante em um SGBD é a de garantir a integridade dos dados consultados através de um controle de concorrênca. Isto significa, por exemplo, que um dado que está sendo alterado por uma transação *A*, não pode ser acessado por uma transação *B*, até o encerramento da primeira transação. Este controle é feito através de bloqueios que são geridos por este módulo, que também é disparado pelo Mecanismo de Execução.

- Gerenciador dos Métodos de Acesso: neste nível o banco de dados é visto como um conjunto de páginas ou registros. Não há a distinção se a informação desejada está em disco ou em memória.

- Gerenciador de *Buffer*: os dados de um banco de dados, por questões de desempenho, podem estar em meio físico permanente (disco) ou volátil (memória), os chamados *buffers*. É este módulo que é responsável por gerir estes *buffers*, que ao ser consultado sobre determinado dado, deve decidir se possui o dado, ou ainda se deve ler ou gravar no disco.

- Gerenciador de Armazenamento: uma informação somente está segura quando gravada em meio físico, o disco. Por questões de segurança e desempenho o armazenamento dos dados em arquivos é feita por este módulo.

- Arquivos: sejam arquivos de índices ou dados, estes são a última instância do banco de dados. É a representação final do que se está armazenando.

2.5.2 Classificações de um SGBD

Os SGBDs podem ser classificados de acordo com o modelo de dados suportado, pelo número de usuários atendidos, pelo tipo de aplicação e pela quantidade de *sites* (locais) de armazenamento (ELMASRI; NAVATHE, 2005).

Ainda de acordo com (ELMASRI; NAVATHE, 2005), quanto ao modelo de dados suportado, o mais utilizado é o **relacional**. Existem implementações de SGBD que suportam dados de **objetos**, mas com uma utilização pequena. Já sistemas mais antigos são suportados por banco de dados **hierárquicos** e de **rede**. Um modelo híbrido denominado **objeto-relacional** é o que se tem como a evolução do **relacional**.

Referente ao número de usuários atendidos pelo SGBD há duas categorias: **Monousuários** - para um único usuário; e **Multiusuários** - que atendem múltiplos usuários simultaneamente.

A classificação quanto ao tipo de aplicação suportada é:

- Tradicionais: manipulam basicamente dados numéricos e textuais.

- Multimídia: podendo manipular imagens, vídeos e sons.

- Informações Geográficas: capazes de armazenar e analisar mapas.

- *Data Warehouses*: para processamento analítico de informações de maneira *on-line*, sendo utilizados para a tomada de decisões.

- Banco de Dados Ativos e Tempo Real: para controle de processos industriais e de produção.

E por fim, com relação ao número de *sites*, tem-se:

- Centralizado: quando os dados e o SGBD estão armazenados em um único local.
- Distribuído: tanto os dados quanto o *software* do SGBD podem estar distribuídos em mais de um local, conectados pela rede. Neste modelo há necessidade de homogeneidade dos *softwares* do SGBD.
- Confederado: tendência atual que consiste de um sistema de múltiplos banco de dados independentes, onde os SGBDs participantes são fracamente acoplados. Podem ser heterogêneos, tanto em dados quanto em *software* do SGBD.

Os SGBDs são uma categoria de *software* muito importante dentro do contexto da Tecnologia da Informação (TI). Com base no que foi descrito, nota-se que os SGBDs evoluiram e evoluem a fim de atender às necessidades que são geradas pela evolução tecnológica e cultural. É dentro deste contexto de evolução que este trabalho se insere.

3 ATUALIZAÇÃO DINÂMICA DE SOFTWARE EM SGBDS COM SUPORTE DO MODELO DE COMPONENTES

Neste capítulo é apresentada a proposta para atualizar o SGBD sem que o seu serviço seja indisponibilizado aos usuários, utilizando técnicas de ADS e com suporte do modelo de componentes de *software*. É proposta uma arquitetura, seu modelo conceitual e as limitações a serem observadas na solução. Será detalhado o Gerenciador de Atualizações (GA), que é um novo componente a ser incorporado à arquitetura. Também serão abordadas características e conceitos do modelo de componentes FRACTAL, que foi utilizado na confecção do protótipo de validação.

3.1 Modelo de SGBD em Componentes de *Software*

Neste trabalho é proposta a construção de um SGBD no modelo de componentes de *software*, um tipo de construção inexistente. Apesar de os SGBDs atuais, em sua maioria, serem construídos em um modelo orientado a processos, já existem implementações construídas no paradigma de orientação a objetos (OO), como o HyperSQL[1]. Estas implementações, se conjugadas com *frameworks* para construção de sistemas com o conceito de componentes de *software*, podem chegar a uma solução compatível ao que se propõe neste trabalho. Para tanto, possivelmente, se passará pela reescrita do SGBD, de tal forma que os objetos que implementam seus módulos serão encapsulados em componentes.

Na seção 2.5 foi apresentada a estrutura básica de um SGBD. Elmasri e Navathe (2005) denominam cada uma destas estruturas de *módulo componente do SGBD*. Então, aplicar a abstração de componentes de *software* sobre a estrutura do SGBD, encapsulando cada módulo em um componente é um caminho natural e o que se propõe. Assim, uma vez que o módulo está encapsulado em um componente, operações de controle pré-definidas podem ser utilizadas para substituir os componentes de *software*.

Este processo de substituição dos componentes necessita de gerenciamento. Um novo componente de *software*, o Gerenciador de Atualizações (GA), é incorporado à arquitetura básica, com o intuito de realizar o gerenciamento do processo de ADS. O componente GA será detalhado na seção 3.3.

[1] Disponível em `http://hsqldb.org`

3.2 Restrições da Solução

Um dos erros mais comuns dos iniciantes na pesquisa científica é tornar o seu objeto de estudo grande demais (DANTON, 2002). Para mitigar o problema, neste trabalho optou-se por ter as seguintes restrições:

1. Atualização de SGBD central de único nó: como trata-se de uma proposta que passou por uma validação, fez-se a opção por um modelo mais simples de SGBD. Como mencionado na seção 3.1 a proposta é de construção do SGBD em um modelo inédito.

2. Obrigatoriedade de manutenção da interface dos componentes: significa que o novo componente deve ser capaz de prover os mesmos serviços do componente atual. Esta restrição não invalida este trabalho, uma vez que ela também foi utilizada no trabalho de Wang et al. (2006) e porque muitas das atualizações de *software* são para corrigir pequenos erros (*bugs*).

3. Não há transferência de estado na substituição do componente: isto significa que o componente para ser substituído necessariamente não estará em uso. Esta restrição é a mesma existente em outros modelos de ADS, como a do PODUS (SEGAL; FRIEDER, 1993).

3.3 Gerenciador de Atualizações (GA)

O Gerenciador de Atualizações (GA) é um componente de *software* incorporado na arquitetura tradicional do SGBD, que terá a responsabilidade de fazer todo o controle da ADS.

Mesmo quando tratando de ADS pode-se afirmar que existem duas formas para a realização das atualizações: manual ou automática. Em uma implementação com atualização manual, um operador irá fornecer os elementos necessários e dará início ao procedimento. Já com uma implementação automatizada, o procedimento é disparado automaticamente por um evento preestabelecido. Preconiza-se que o GA possa trabalhar desta forma.

Com relação às atribuições do GA se pode afirmar que é capaz de:

- Controlar a versão dos componentes.
- Identificar o momento de realizar as substituições.
- Garantir a integridade estrutural do sistema.
- Controlar a demanda das requisições ao sistema.

- Realizar a substituição de um subcomponente.

Devido à quantidade e à complexidade das atividades sob responsabilidade do GA, é natural que ele seja concebido na forma de um componente composto por outros subcomponentes. Cada subcomponente é responsável pela realização de uma das suas atribuições. Na Figura 3.1 se pode observar os subcomponentes do GA e as suas relações. O subcomponente mais importante é o *Atualizador do Sistema*, pois é ele que faz as substituições dos componentes. Para isto, ele se vale das informações geradas pelos demais subcomponentes.

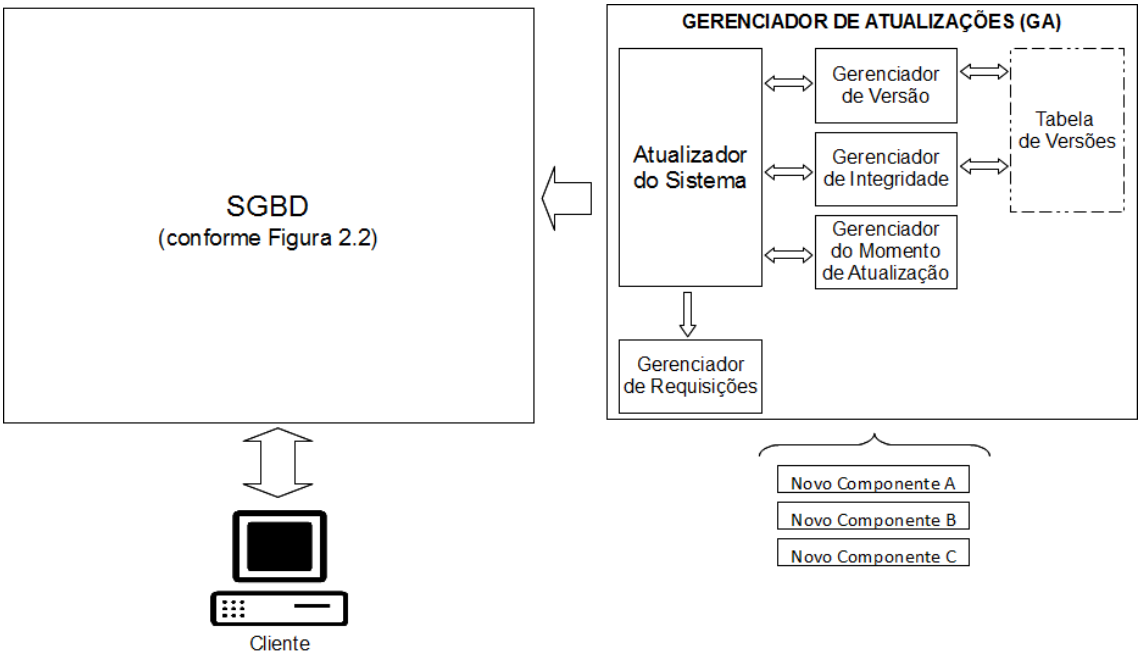

Figura 3.1: Arquitetura do GA

Os subcomponentes Gerenciador de Versões (seção 3.3.1), Gerenciador do Momento de Atualização (seção 3.3.2), Gerenciador de Integridade (seção 3.3.3), Gerenciador de Requisições (seção 3.3.4) e o Atualizador do Sistema (seção 3.3.5) serão abordados na sequência.

3.3.1 Gerenciador de Versões

O **Gerenciador de Versões** dos componentes é responsável por uma atividade fundamental para o funcionamento adequado da ADS. De acordo com Collins-Sussman et al. (2004), o controle de versão é a arte de gerenciar as mudanças das informações. Os controladores de versões tradicionais trabalham em uma arquitetura cliente-servidor. Existe um repositório centralizado onde são armazenadas os arquivos com informações que se desejam o controle através do tempo. Os clientes, identificados como **área de trabalho**, obtém estes arquivos a partir do

repositório (*checkout*) ou apenas o que foi alterado (*update*), podendo promover alterações e depois submeter (*commit*) esta nova versão. Cada vez que é feita uma submissão é criado uma nova revisão. A última revisão é conhecida como versão *HEAD*.

No modelo proposto, o SGBD em execução, mais um *log* de componentes que foram substituídos representam uma variação particionada do repositório (Figura 3.2). Ou seja, o próprio SGBD representa a versão *HEAD*.

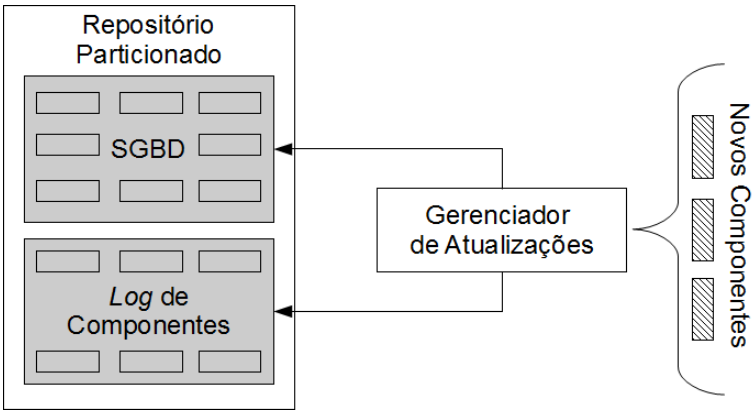

Figura 3.2: Modelo de repositório particionado da solução

Para o adequado controle das versões, cada componente deve ser capaz de se identificar unicamente ao GA. Esta identificação deve atender dois aspectos:

- **Tipo de Componente:** o componente deve informar ao GA a qual tipo de componente ele se refere.
- **Versão do Componente:** a indicação da versão deve obedecer a uma ordem cronológica de finalização do componente[2].

Internamente, o GA deve manter uma tabela com as versões atuais do sistema, como por exemplo a da Figura 3.3. Esta tabela deve conter o tipo de componente, a sua versão, qual é a interface que este componente deve satisfazer e a indicação de qual componente[3]. Ela tem três aplicações principais:

- Servir de base para recuperação do sistema, em caso de queda.
- Facilitar o processo de atualização dos componentes, já que mantém a versão atual de cada componente.
- Servir de suporte na validação da interface implementada pelo novo componente.

[2]O instante que ele é liberado para a atualização no sistema.
[3]Considerando a implementação em *Java*, trata-se do arquivo *.class* que implementa o componente.

Tipo de Componente	Versão	Componente	Interface
Parser	10/03/2011 12:03:56	parser.class	IParser
Optimizer	10/03/2011 09:45:31	optimizer.class	IOptimizer
ExecutionPlan	07/04/2011 23:10:29	executionPlan2.class	IExecutionPlan
OperatorEvaluator	10/03/2011 16:43:22	operatorEvaluator.class	IOperatorEvaluator
ExecutionEngine	15/03/2011 07:33:20	executionEngine.class	IExecutionEngine
TransactionManager	14/03/2011 03:44:12	transactionManager.class	ITransactionManager
RecoveryManager	09/03/2011 20:24:15	recoveryManager.class	IRecoveryManager
LockManager	22/03/2011 01:08:39	lockManager.class	ILockManager
FileAccessAndMethods	10/03/2011 04:17:46	fileAccessAndMethods.class	IFileAccessAndMethods
BufferManager	07/03/2011 11:05:00	bufferManager.class	IBufferManager
StorageManager	06/03/2011 15:15:32	storageManager.class	IStorageManager

Figura 3.3: Exemplo de tabela de versões dos componentes do SGBD

3.3.2 Gerenciador do Momento da Atualização

Um dos objetivos de ADS é evitar a indisponibilização do serviço. Mas só isto não é suficiente. Deve-se evitar também a queda de desempenho na realização das tarefas, porque desempenho insatisfatório parece inoperância do serviço para o usuário. Desta forma, o momento em que a atualização deverá acontecer é bastante importante. Se o sistema estiver com grande demanda, a sobrecarga gerada pela atualização, mesmo que pequena, pode ser o acréscimo suficiente para comprometer o desempenho do sistema. O subcomponente Gerenciador do Momento de Atualização (GMA) é o mecanismo capaz de detectar o momento adequado para a realização da atualização.

Para prever o momento adequado da atualização, o GMA deve se valer da carga atual do sistema. Informações tais como quantas transações estão ativas no momento, o custo de cada uma delas e o uso da CPU parecem ser suficientes para se chegar a uma medida de carga do sistema. Comparando esta medida com um parâmetro previamente definido pelo DBA e que está acessível ao GMA, este é capaz de decidir se a atualização pode acontecer neste instante, fazendo esta indicação ao Atualizador do Sistema.

Essa é uma forma de implementação mais simples e fácil, mas não a ideal. Sugere-se para a capacidade de identificar a carga do sistema e de julgar o momento da atualização a utilização de características de uma implementação de sistema com substituição autonômica (KEPHART; CHESS, 2003) de componentes, onde o administrador pode definir políticas ou regras para a realização das atualizações. Neste tipo de solução, o sistema irá se utilizar de dados histórí-

cos para construir uma regra de quando é o momento ideal da atualização. Para Oreizy et al. (1999), um *software* capaz de se auto atualizar deve ter políticas claras de funcionamento. Estas políticas devem considerar o momento da atualização, o grau de autonomia (quanto de interferência manual de operadores no processo de atualização), o tipo de atualização (se refere a alteração estrutural do *software* e especificamente quando tratando de componentes de *software* se é permitida a alteração da interface ou não) e a frequência das atualizações.

3.3.3 Gerenciador de Integridade

Como definido na seção 2.3, componentes de *software* possuem uma interface bem definida. A interface é uma coleção de operações que representa os serviços disponibilizados. Assim, para garantir a integridade estrutural do sistema, é suficiente e necessário que o novo componente tenha a mesma interface do componente que está sendo substituído.

Cabe ao Gerenciador de Integridade (GI) fazer esta verificação e comunicar ao Atualizador do Sistema. Para tanto ele consulta a tabela de versões (Figura 3.3) para identificar qual é a interface que o componente deve implementar. Então o GI verifica se a interface está implementada na íntegra pelo novo componente.

É importante salientar que é uma validação de integridade estrutural e não comportamental do componente. Isto significa que o novo componente tem as mesmas operações do antigo. Mas não há garantias quanto ao seu funcionamento, ou seja, quanto aos resultados que serão produzidos por suas operações.

3.3.4 Gerenciador de Requisições

Para Wang et al. (2006), o cumprimento da premissa de não indisponibilização do serviço para o usuário final significa que nenhuma requisição pode ser recusada. Dessa forma, o Gerenciador de Requisições (GR) precisa estar apto a controlar as requisições de manipulação de dados encaminhadas ao sistema. Isto porque se ele não for capaz de manter este controle e interferir diretamente no repasse destas requisições, o que acontece se chegar uma requisição durante o processo de atualização? Ao iniciar o processo de atualização de um componente ele deverá enfileirar as novas requisições que chegarem ao sistema. Isto significa bloquear as requisições durante a atualização, como preconizado por Wang et al. (2006). No final, o GR repassa esta fila de requisições bloqueadas ao componente apropriado do SGBD.

3.3.5 Atualizador do Sistema

O processo de substituição de um subcomponente é de responsabilidade do Atualizador do Sistema (AS). Em linhas gerais, consiste em identificar que um novo subcomponente está disponível para atualização, solicitar ao Gerenciador de Versão e ao Gerenciador de Integridade para verificar se este componente é válido e na sequência proceder as operações de controle necessárias para a substituição.

Na Figura 3.4 o processo é apresentado de uma forma algorítmica. É uma operação permantente, ou seja, enquanto o sistema estiver em operação, substituições de subcomponentes poderão ocorrer (Passo 1). Desta forma o AS deve continuamente estar verificando uma *fila* de novos componentes (Passo 2). A partir da existência de um novo componente nesta *fila*, a substituição do subcomponente é iniciada (Passo 3). O novo componente então é identificado e seu tipo é validado (Passos 4 e 5). Isto ainda é insuficiente para proceder a substituição do subcomponente. Este novo componente deve ser mais *novo* que o atual (Passos 6 e 7). Estas operações são de responsabilidade do pelo Gerenciador de Versões. Ainda o novo componente deve implementar todas as operações da *interface* (Passos 8 e 9), o que é feito pelo Gerenciador de Integridade. Na sequência, o AS deve solicitar ao Gerenciador do Momento de Atualização a decisão sobre momento da atualização, isto é, se naquele instante é possível ou não. Como destacado na seção 3.3.2, a substituição do subcomponente somente deve acontecer se o impacto sobre o sistema não ferir a política de desempenho. Isto significa que em momentos de maior carga do sistema, a operação não deve ser realizada. O Gerenciador do Momento de Atualização então avalia o sistema (Passo 10) e decide pela substituição do subcomponente, notificando o AS da situação (Passo 11). O AS inicia a substituição do subcomponente parando o componente raiz (Passo 12). Isto significa que a partir deste instante, todos os subcomponentes não executarão mais as suas funções normais. Segue-se então a sua desconexão (Passo 13), o que vai liberá-lo para a troca. Então é removido (Passo 14), ou seja, este componente deixa de ser um subcomponente da arquitetura. O novo componente então é adicionado (Passo 15). Neste instante ainda não está apto a funcionar, pois precisar ser informado de quais são suas ligações. Conectado o novo componente (Passo 16), então o componente raiz pode ser iniciado (Passo 17). Faltam apenas operações de controle como a atualização da tabela de controle das versões que é feita pelo Gerenciador de Versão (seção 3.3.1 / Figura 3.3) e a movimentação do componente para o *log* de componentes (Passos 18 e 19), feita pelo próprio AS.

Um melhoramento que pode ser feito na proposta aqui apresentada para o GA é a possibi-

```
Enquanto o sistema estiver ativo Faca
  Ler a fila novos componentes
  Se existe um novo componente para substituir Entao
    Ler o tipo do componente
    Se é um componente válido Entao
      Ler a versão do componente
      Se o componente é mais novo Entao
        Validar a interface do novo componente
        Se o novo componente tem a interface válida Entao
            Avaliar carga do sistema
            Se o sistema pode ser atualizado Entao
              Parar o componente raiz
              Desconectar o subcomponente a ser substituído
              Remover o subcomponente
              Adicionar o novo subcomponente
              Conectar o novo subcomponente aos demais
              Iniciar o componente raiz
              Mover o componente antigo para o log
              Atualizar a tabela de versões
            Senao
              IrPara o Passo 10
            Fim Se
        Fim Se
      Fim Se
    Fim Se
  Fim Se
Fim Enquanto
```

Figura 3.4: Passos para substituição de um subcomponente

lidade de reverter uma atualização. Para isto, além de guardar os componentes antigos (*log* de componente), ele deverá manter uma tabela com a ordem das atualizações.

Outra melhoria é a necessidade deste processo ter características ACID. FRACTAL é um modelo de componentes em que se pode incluir estas características quando da reconfiguração dos componentes, como já demonstrado no trabalho de Léger et al. (2007).

3.4 Descrevendo um SGBD no Modelo de Componentes FRACTAL

Com o uso de FRACTAL como suporte para descrever o SGBD baseado no modelo de componentes, há pelo menos duas opções para isto, através de FRACTAL ADL ou FRACTAL API.

3.4.1 FRACTAL ADL

FRACTAL ADL consiste de um arquivo XML para descrever os componentes, determinando seus tipos, interfaces, ligações, implementações e hierarquia. Na Figura 3.5 é apresentado um exemplo da descrição de um componente proposto neste trabalho.

Os tipos são a própria definição dos componentes, como eles são e o que fazem. Estão destacados na Figura 3.5 com o destaque (A).

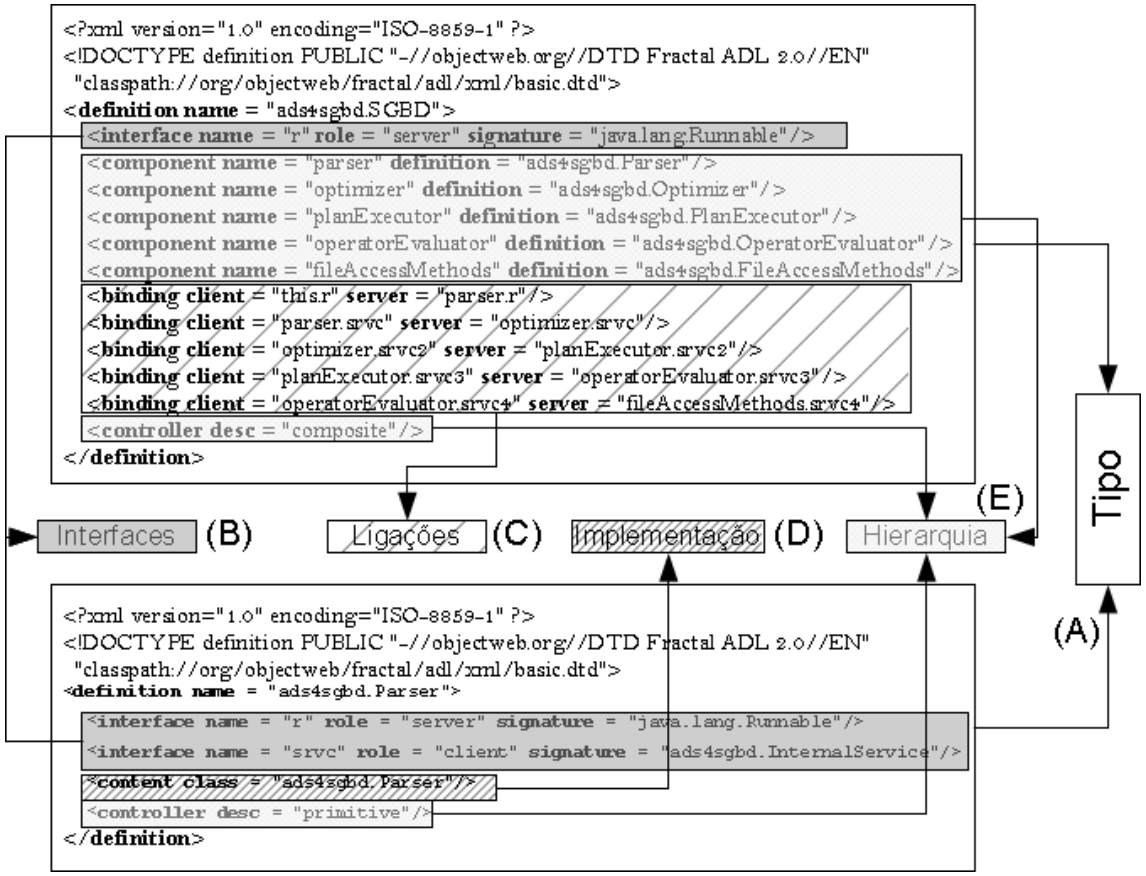

Figura 3.5: Componentes descritos com Fractal ADL

As *interfaces* representam os serviços requisitados e disponibilizados pelos componentes. Observando a Figura 3.5 com o destaque (B), a cláusula ***role*** distingue as *interfaces* requeridas (***client***) e as providas (***server***).

As ligações são identificadas pela cláusula ***binding***, destaque (C) da Figura 3.5. Observa-se também que uma *interface* cliente de um componente é ligada a uma *interface* servidora de outro componente.

As Implementações são identificadas pela cláusula ***content class*** estando na Figura 3.5 com o destaque (D) e tem a função de determinar a classe (no caso em *Java*) de implementação do componente.

A hierarquia dos componentes na arquitetura é definida pelas cláusulas ***component name*** e ***controller desc***, podendo ser visualizada na Figura 3.5 destaque (E).

3.4.2 FRACTAL API

Atualmente estão definidas **FRACTAL APIs** para *Java*, *C* e *OMG IDL*, sendo que a terceira garante a interoperabilidade entre componentes escritos em qualquer linguagem de programa-

ção. As declarações feitas através das APIs são mais fáceis de serem observadas e identificadas. Porém há a necessidade de se fazer as descrições completas, diferente de uma implementão através de uma ADL. Na Figura 3.6 são mostrados trechos de códigos retirados da implementação da solução proposta neste trabalho, com os métodos e componentes mais significativos[4] e que são descritos a seguir:

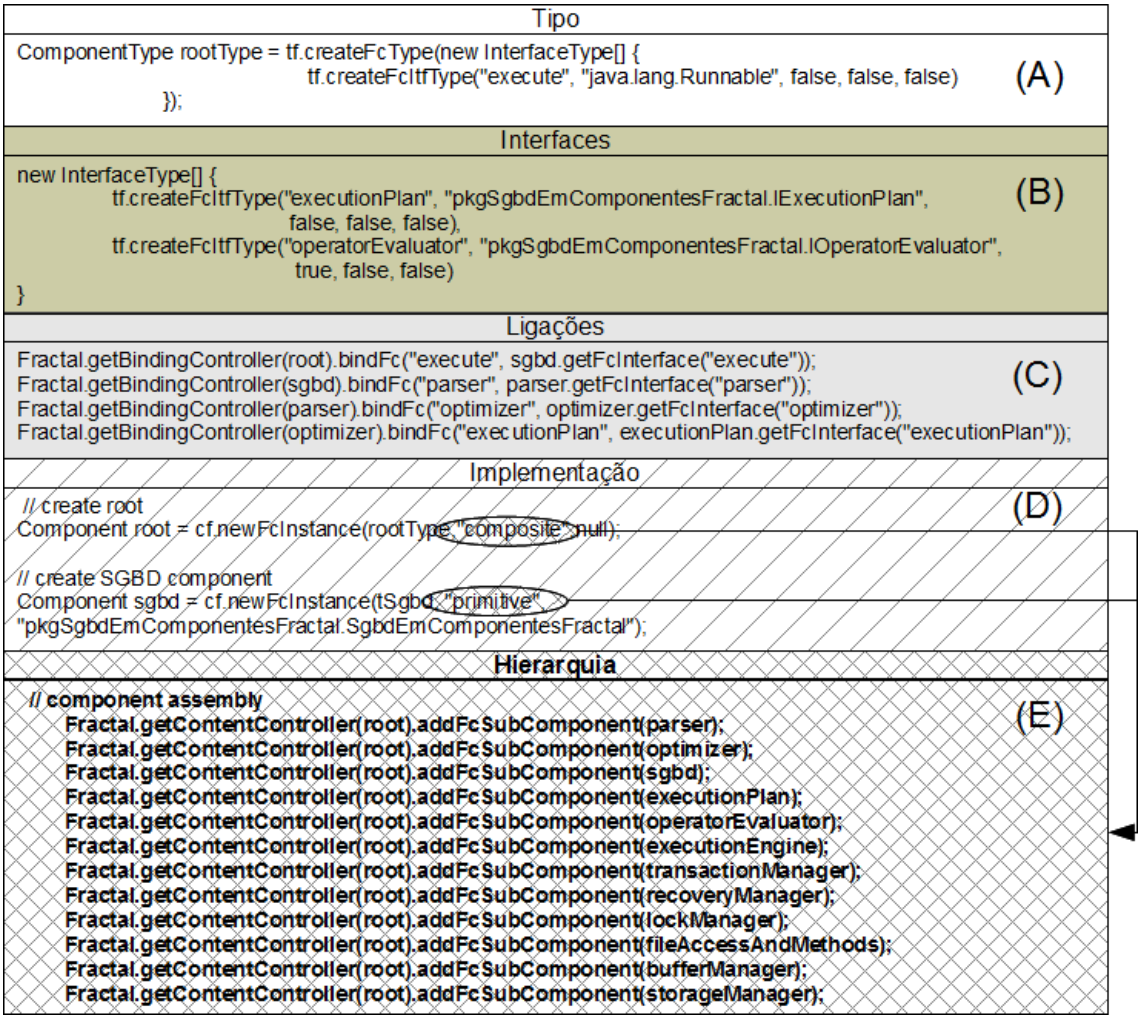

Figura 3.6: Componentes descritos com Fractal API em Java

- Tipos: Na Figura 3.6 - (A) as cláusulas ***ComponentType*** e ***createFcType*** são responsáveis pela definição dos tipos.
- *Interfaces*: Observando a Figura 3.6 - (B), a cláusula ***new InterfaceType*** representa o componente que contém a descrição da *interface* que se está definindo. As *interfaces* providas e requeridas são especificadas através da cláusula ***createFcItfType***. Este método recebe como parâmetros:

[4]Pode-se verificar a definição completa de cada uma destas APIs em `http://fractal.ow2.org/specification/index.html#tth_sEcA`

  - Nome: define o nome da *interface*.
  - Assinatura: relaciona a *interface* do componente com a *interface* na linguagem.
  - Requerida/Cliente ou Provida/Servidora: um parâmetro booleano para identificar se esta *interface* é requerida (*true*) ou provida (*false*).
  - Interface Opcional: mais um parâmetro booleano para determinar se esta é uma *interface* opcional (*true*) ou obrigatória (*false*).
  - Permitir múltiplas *interfaces* iguais: este parâmetro booleano serve para identificar se o componente poderá ter mais de uma *interface* deste tipo.

- Ligações: são identificadas pela cláusula ***bindFc*** (Figura 3.6 - (C)).
- Implementações: cria-se uma nova instância de um componente FRACTAL (3.6 - (D)) com o método ***newFcInstance***, informando os parâmetros tipo de componente, se é um componente primitivo ou composto e a classe de implementação.
- Hierarquia: a cláusula ***addFcSubComponent*** (Figura 3.6 - (E)) associado com o segundo parâmetro do método ***newFcInstance*** definem a hierarquia dos componentes na arquitetura.

4 IMPLEMENTAÇÃO

Foi desenvolvido um protótipo para comprovação da viabilidade técnica da solução proposta. Para sua construção optou-se pela utilização da linguagem *Java* (versão 1.6), com *Julia* (versão 2.5.1) como a implementação de referência de FRACTAL. Adotou-se o ambiente de desenvolvimento e compilação (IDE - *Integrated Development Enviroment*) *IntelliJIDEA*, na sua versão 10.0.3 *Community Edition*, da *JetBrains*[1]. Para facilitar a integração do desenvolvimento do GA (seção 3.3) optou-se pela construção com APIs.

4.1 Modelo de Classes

Procurou-se ser o mais fiel possível à estrutura descrita na seção 2.5 (vide Figura 2.2). Mas dificuldades durante o desenvolvimento impediram alcançar deste objetivo. Destaca-se em especial a impossibilidade de um componente funcionar como *servidor* para mais de um componente. Isto exigiu uma solução de contorno com a simplificação do modelo, como está descrito na Figura 4.1. Dessa forma houve uma sequencialização das atividades, ou seja, um componente só requisita serviço de um componente e provê serviço para um componente e um passo de cada vez.

[1] `http://www.jetbrains.com/idea/`

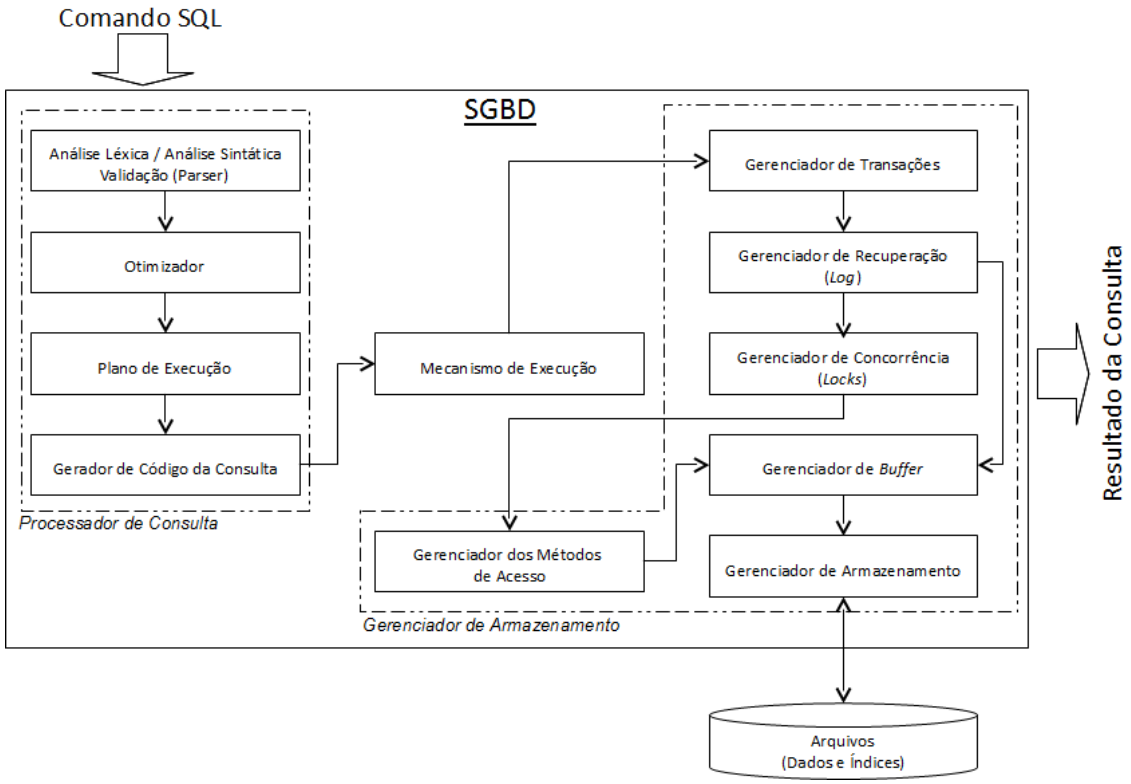

Figura 4.1: Esquema do protótipo implementado em FRACTAL

O modelo de classes da implementação está no diagrama UML (*Unified Modeling Language*) da Figura 4.2. A classe *Componente* (Figura 4.2 - (B)) é a classe ancestral de todos os subcomponentes do SGBD. *Control* (Figura 4.2 - (A)) é a implementação simplificada do Gerenciador de Atualizações. É dita simplificada porque realiza apenas algumas das funções descritas na seção 3.3. A classe *SgbdEmComponentesFractal* (4.2 - (D)) funciona como porta de entrada das requisições ao SGBD. As classes *Parser, Optimizer, ExecutionPlan, OperatorEvaluator, ExecutionEngine, TransactionManager, RecoveryManager, LockManager, FileAccessAndMethods, BufferManager* e *StorageManager* (4.2 - (E)) representam os módulos componentes do SGBD. Nesta implementação, com o propósito de simplificação, todos os componentes executam a mesma atividade. Consiste do processamento feito pela classe *Process* (Figura 4.2 - (C)). Este processamento é a geração de uma *string* com a identificação do componente que realizou a atividade, mais um identificador único da instância e um *timestamp* do processamento.

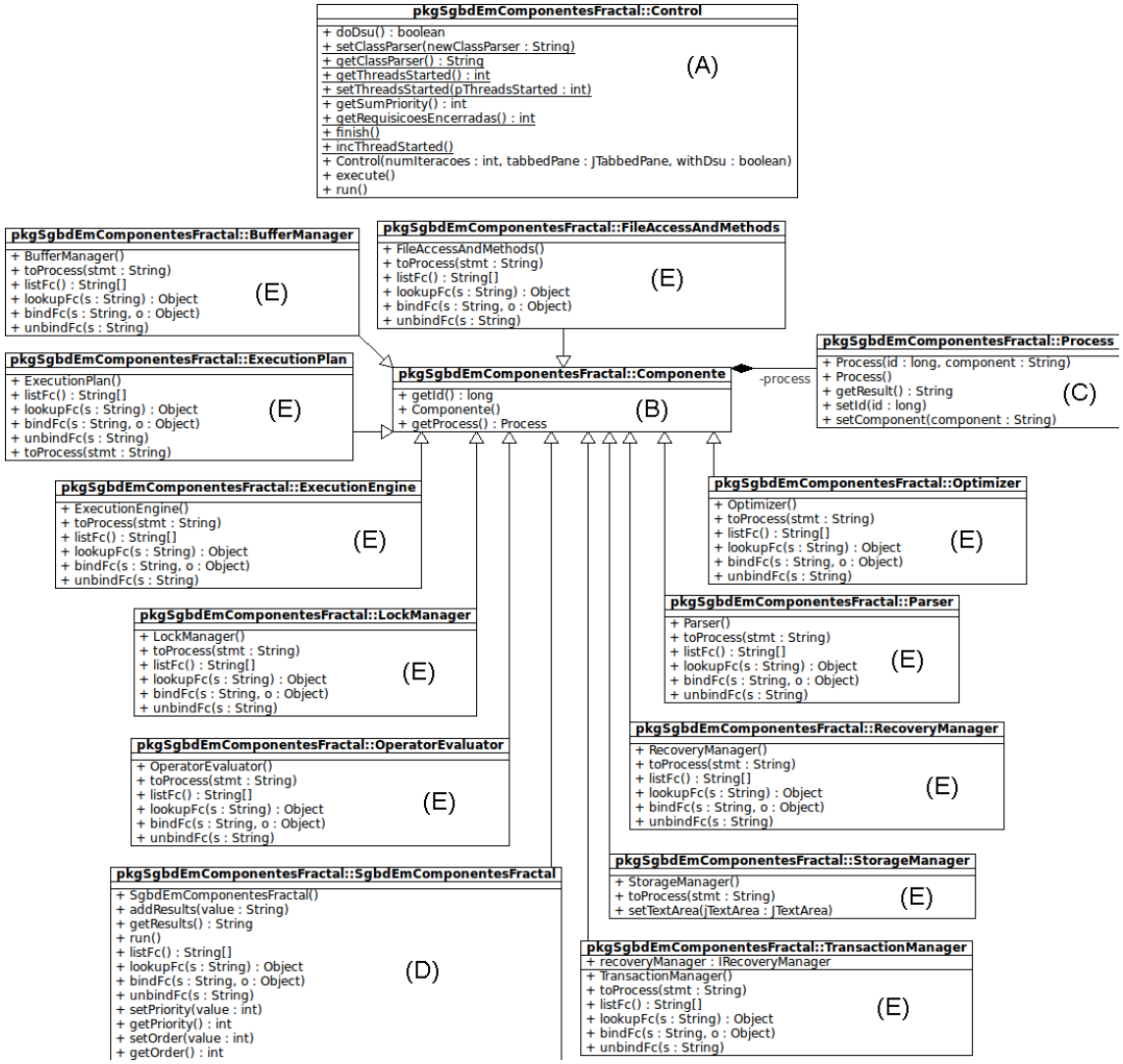

Figura 4.2: Diagrama de classes UML da implementação

4.2 Diagrama de Sequência

Para Quatrani (1999), um diagrama de sequência mostra a interação entre os objetos em uma determinada sequência no tempo.

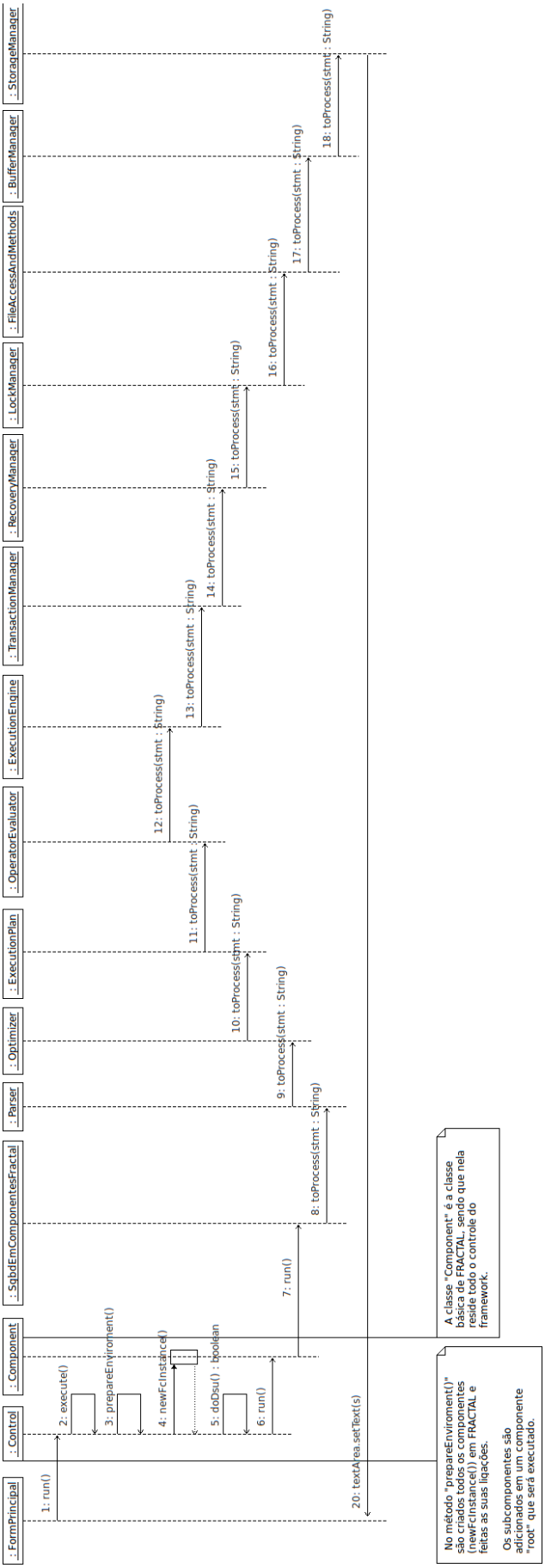

Figura 4.3: Diagrama de sequência UML da implementação

A Figura 4.3 apresenta o diagrama de sequência da implementação. O ponto inicial é uma tela (classe *FormPrincipal*) a partir da qual é disparada a classe *Control*. Para esta classe são passados dois parâmetros: número de requisições a serem processadas e um indicador para realização ou não de ADS durante o processamento. Para cada requisição é aberta uma *thread* que realizará todo o processamento. Ainda na classe *Control* é preparado todo o ambiente de execução FRACTAL (seção 4.3), com o método *prepareEnviroment()*. Para FRACTAL todos os componentes são instâncias de *Component*, criadas com o método *newFcInstance()*. Detalhes da criação do ambiente de execução FRACTAL serão tratados na seção 4.3.

4.3 Criação do Ambiente em FRACTAL

A implementação *Julia* é um ambiente de execução que está completamente de acordo com as especificações de FRACTAL. Para que uma aplicação execute sobre este ambiente é necessário a preparação do mesmo. A Figura 4.3 mostra trechos do método *prepareEnviroment()* da classe *Control*, com as principais funções utilizadas para criação do ambiente e dos componentes FRACTAL. Na linha 4, o método *getBootstrapComponent* cria o componente inicial de todo o ambiente. A linha 6 cria a fábrica de tipos (*getTypeFactory*). Os tipos são criados com o método *createFcType* (linhas 9 a 27). A fábrica dos demais componentes é criada com o método *getGenericFactor* observado na linha 30. Com o método *newFcInstance* são criadas novas instâncias de componentes FRACTAL (linhas 32 a 43). A composição dos componentes (ou adição de um subcomponente a um componente composto) é feito com o método *addFcSubComponent*, presentes entre as linhas 47 e 51. A ligação dos subcompoentes entre si (linhas 55 a 60) é feito com o método *bindFc*. O ambiente criado é colocado em execução com o método *startFc* (linha 63).

```
1  private void prepareEnviroment() throws Exception  {
2
3      // criar os componentes do SGBD
4      boot = org.objectweb.fractal.util.Fractal.getBootstrapComponent();
5      // factory dos tipos
6      tf = org.objectweb.fractal.util.Fractal.getTypeFactory(boot);
7
8      // type of root component
9      rootType = tf.createFcType(new InterfaceType[] {
10      tf.createFcItfType("execute", "java.lang.Runnable", false, false, false)
11     });
12
13     // definindo os tipos / interfaces implementadas
14     // type SgbdEmComponentes
15     tSgbd = tf.createFcType(new InterfaceType[] {
16      tf.createFcItfType("execute", "java.lang.Runnable", false, false, false),
17      tf.createFcItfType("parser", "pkgSgbdEmComponentesFractal.IParser",true,false,
            false),
18      tf.createFcItfType("sgbdAttributes", "pkgSgbdEmComponentesFractal.SGBDAttributes",
            false, false, false)
19     });
20     ...
21     ...
22     ...
23     // type Storage Manager
24     tStorageManager = tf.createFcType(new InterfaceType[] {
25      tf.createFcItfType("storageManager", "pkgSgbdEmComponentesFractal.IStorageManager"
            , false, false, false),
26      tf.createFcItfType("textArea", "pkgSgbdEmComponentesFractal.ITextArea", false,
            false, false)
27     });
28
29     // factory dos componentes
30     cf = org.objectweb.fractal.util.Fractal.getGenericFactory(boot);
31     // create root
32     root = cf.newFcInstance(rootType,"composite",null);
33
34     // criando todos os componentes
35     // create SGBD component
36     sgbd = cf.newFcInstance(tSgbd, "primitive", "pkgSgbdEmComponentesFractal.
           SgbdEmComponentesFractal");
37     // create Parser
38     parser = cf.newFcInstance(tParser, "primitive", Control.getClassParser());
39     ...
40     ...
41     ...
42     // create StorageManager
43     storageManager = cf.newFcInstance(tStorageManager, "primitive", "
           pkgSgbdEmComponentesFractal.StorageManager");
44
45     // adicionando os sub-componentes
46     // component assembly
47     org.objectweb.fractal.util.Fractal.getContentController(root).addFcSubComponent(
           parser);
48     ...
49     ...
50     ...
51     org.objectweb.fractal.util.Fractal.getContentController(root).addFcSubComponent(
           storageManager);
52
53
54     // bindings - fazendo as ligações
55     org.objectweb.fractal.util.Fractal.getBindingController(root).bindFc("execute",
           sgbd.getFcInterface("execute"));
56     org.objectweb.fractal.util.Fractal.getBindingController(sgbd).bindFc("parser",
           parser.getFcInterface("parser"));
57     ...
58     ...
59     ...
60     org.objectweb.fractal.util.Fractal.getBindingController(bufferManager).bindFc("
           storageManager", storageManager.getFcInterface("storageManager"));
61
62     // start root component - iniciando o componente raiz
63     org.objectweb.fractal.util.Fractal.getLifeCycleController(root).startFc();
64 }
```

Figura 4.4: Trechos do método *prepareEnviroment()* da classe *Control*

4.4 Aspectos Gerais da Implementação

Como a implementação deste trabalho tem objetivos de comprovação e experimentação de uma proposta, havia necessidade de parâmetros para comparação. Neste sentido se optou pela implementação de dois protótipos, um construído sob o paradigma de OO e outro baseado em componentes de *software*. Os dois protótipos tem funcionamento análogos, realizando as mesmas atividades de processamento.

Tradicionalmente os aplicativos que fazem uso do SGBDs são desenvolvidos em uma arquitetura cliente/servidor. No desenvolvimento do protótipo de validação e experimentação esta arquitetura foi respeitada. Para tanto foi desenvolvido um aplicativo de entrada (*front end* / Figura 4.5) que tem as seguintes atribuições:

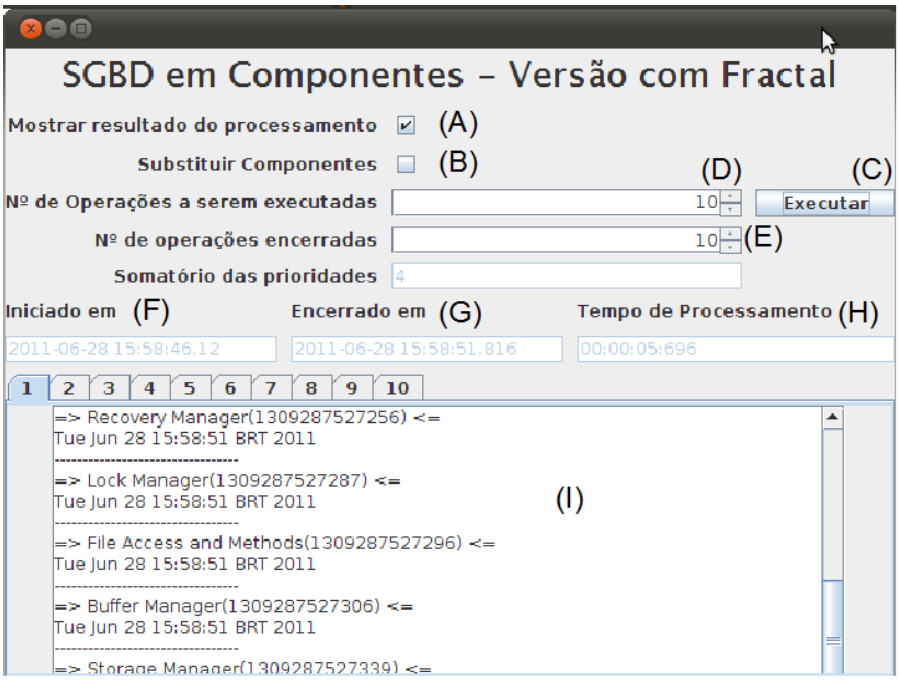

Figura 4.5: Forma de apresentação do protótipo com FRACTAL

- coletar os parâmetros de execução, que são o número de requisições a processar (figura 4.5 - (D)) e o indicador para realização de ADS ou não (Figura 4.5 - (B));
- disparar a execução do protótipo através do botão *Executar* (Figura 4.5 - (C));
- apresentar os resultados da execução, sendo os produtos finais como seguem:
  - número de operações encerradas (Figura 4.5 - (E));
  - tempos de execução, com o tempo inicial (Figura 4.5 - (F)), o tempo final (Figura 4.5 - (G)) e o tempo total de processamento (Figura 4.5 - (H));

– resultados produzidos pelos componentes do SGBD que são apresentados em uma espécie de console (Figura 4.5 - (I)). Para cada *thread* em execução é criada um console (abas numeradas de 1 a *n*). Estes resultados são apresentados somente se o usuário marcar a opção correspondente (Figura 4.5 - (A)).

5 TESTES E AVALIAÇÃO DE RESULTADOS

Este capítulo apresenta a avaliação do protótipo implementado e consequentemente da arquitetura proposta como solução. A seção 5.1 apresenta o ambiente de execução de testes utilizados durante o processo de avaliação e os cenários considerados. Na seção 5.2 tem-se a avaliação experimental do protótipo e os resultados obtidos. A seção 5.3 relaciona propriedades de qualidade de *software* que a solução cumpre e como foram comprovadas.

5.1 O Ambiente e os Cenários da Avaliação

O desenvolvimento e testes dos protótipos foram realizados em uma máquina virtual *Oracle Virtual Box* versão 4.0.4, com sistema operacional *Linux Ubuntu* versão 10.10, sendo a memória base de 512 MB. Este ambiente rodava sobre uma máquina real com processador *Intel Core 2 Duo T8300*, 2 GB de memória principal e sistema operacional *Windows XP Professional* versão 2002 SP 2.

A avaliação considerou três cenários distintos, a saber:

1. Execução de um protótipo construído de forma tradicional (OO), portanto sem FRACTAL.

2. Execução de um protótipo construído na perspectiva de componentes de *software* baseado em FRACTAL, mas sem a realização de ADS.

3. Execução de um protótipo construído na perspectiva de componentes de *software* baseado em FRACTAL, mas com a realização de ADS.

A opção por estes cenários objetivava fazer avaliações comparativas de desempenho entre os cenários 1 e 2 e entre os cenários 2 e 3. O intuito é identificar se existe e qual é a perda de desempenho entre as opções.

5.2 Avaliação Experimental

Uma avaliação experimental objetiva observar fenômenos manipulando as variáveis envolvidas, para saber as causas e o efeitos de como o evento ocorre (DANTON, 2002). Antes de determinar qual o tipo de avaliação seria utilizada neste trabalho, foram identificadas duas propostas de avaliações com base em experimentação sobre ADS:

- Gharaibeh *et al.* (2009) propuseram uma avaliação quantitativa de custo/benefício para ADS. A proposta consiste em avaliar se o custo extra adicionado ao sistema para suportar ADS é menor do que uma interrupção para atualização *off line*. Somente assim, vale a pena utilizar ADS para atualizações de sistemas não críticos.

- Salmi *et al.* (2009) avaliam a variação do tempo de resposta e do uso de memória durante procedimentos de reconfiguração do sistema analisado. Concluem que as reconfigurações do sistema não impactam negativamente sobre os dois elementos avaliados. O ambiente experimentado também foi implementado em FRACTAL com a implementação de referência *Julia* para *Java*.

Neste trabalho optou-se pela avaliação apenas de tempo de resposta, mas com três cenários para comparação, conforme descrito na seção 5.1. Os testes realizados consistiram da execução de grupos de requisições e obtenção dos tempos gastos.

Os grupos ou quantidades (*NumRequisicoes*) definidos para os testes foram: 100, 500, 1.000, 1.500, 2.000, 2.500 e 3.000 requisições. O ambiente de testes não suportou uma quantidade maior de requisições. Ocorre *estouro* de memória por limitação do *hardware*. Para obtenção de resultados mais íntegros, cada conjunto de requisições foi executado 5 vezes.

A medida apurada foi o número médio de requisições processadas em 1 segundo (*NumRequisicoesPorSeg*). Esta medida é uma relação da quantidade de requisições processada pela média aritmética simples dos tempos das 5 execuções (*TempoMedio*). Assim, tem-se que:

$$NumRequisicoesPorSeg = \frac{NumRequisicoes}{TempoMedio} \tag{5.1}$$

Na Tabela 5.1 são apresentados os resultados obtidos nos três cenários e nas quantidades de requisições executadas.

Número de Requisições	**Protótipo SEM FRACTAL**	**Protótipo COM FRACTAL SEM ADS**	**Protótipo COM FRACTAL COM ADS**
100	17,105	13,478	13,762
500	60,474	43,564	43,658
1.000	92,095	57,283	59,525
1.500	115,632	70,783	71,234
2.000	120,620	74,090	73,958
2.500	120,020	80,334	77,912
3.000	119,100	85,115	80,935

Tabela 5.1: Comparativo do número de requisições processadas em 1 segundo

Fixando o desempenho do protótipo construído sem FRACTAL, observa-se que o desempenho decresce aproximadamente em média **30%**. Para se obter este valor, primeiramente verificou-se o percentual de redução do número de requisições processadas por segundo do protótipo com FRACTAL e sem a realização de ADS. No gráfico da Figura 5.1, a série **FRACTAL sem ADS** mostra os valores obtidos nas execuções para 100, 500, 1.000, 1.500, 2.000, 2.500 e 3.000 requisições. Calculando a média desta variação é obtido o valor de **32,28%**. Na sequência, calculou-se o percentual de variação para a execução com realização de ADS. No mesmo gráfico a série **FRACTAL com ADS** registra os valores obtidos. A média calculada ficou em **32,42%**.

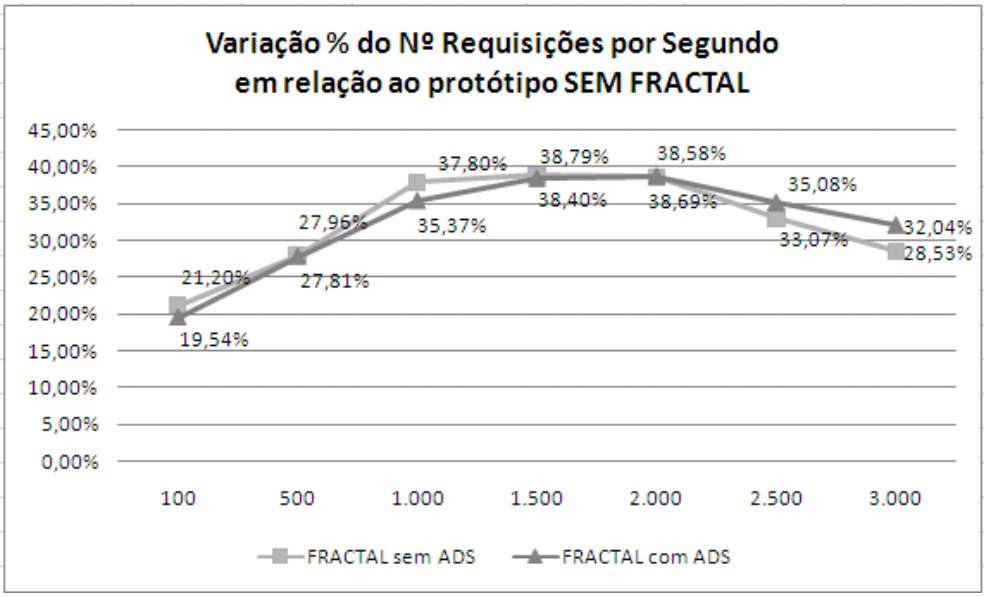

Figura 5.1: Gráfico de variação % do número de requisições por segundo em relação ao protótipo SEM FRACTAL

A indisponibilização do sistema provoca dois tipos de perdas, a saber:

- Diretas e Imediatas: como por exemplo deixar de realizar uma venda em um sistema de *e-commerce*.
- Indiretas e a Longo Prazo: se valendo do mesmo exemplo, o prejuízo de imagem pode provocar a perda em definitivo do cliente. Isto é, o cliente não retorna mais a esta loja virtual por ter tido experiência negativa por indisponibilidade do sistema.

Desta forma, esta degradação é aceitável uma vez que não implica na indisponibilização do sistema. Reforça esta informação o fato de que a ocorrência de ADS pode ser programada para acontecer em momentos de menor carga do sistema, minimizando o impacto sobre o mesmo.

5.3 Propriedades da Solução

De acordo com Rezende (2006), um *software* é medido por muitas razões, entre elas para identificar a qualidade do mesmo. São medidas e métricas de qualidade: corretude, integridade, manutenabilidade e usabilidade. Baseadas nestas métricas, determinou-se duas propriedades a serem cumpridas pela solução: corretude e manutenabilidade.

5.3.1 Corretude

Para Rezende (2006) a corretude determina o grau com que o *software* executa de forma correta o que lhe é exigido. Nesta avaliação quer se determinar a capacidade do protótipo fazer o que se propõe, ou seja, executar as requisições recebidas e em determinado momento realizar ADS. O método utilizado foi o da verificação dos resultados produzidos pelo protótipo desenvolvido com FRACTAL e realizando ADS.

O teste realizado consistiu em executar o protótipo para um determinado número *n* de requisições. Após a execução de *n'*[1] requisições ocorre a ADS. O subcomponente *Parser* é substituído pelo subcomponente *ParserV2* ou o contrário, dependendo de qual deles estava em uso no instante da substituição.

A Figura 5.2-(A) mostra os resultados produzidos pelo processamento da primeira requisição, aba de console número 1, identificado pelo destaque (1). O destaque (2) identifica que esta *thread* executou com o subcomponente *Parser*. Já o destaque (3) mostra o *timestamp* em que o subcomponente *Parser* realizou o seu processamento.

[1] Correspondente a um percentual de *n*.

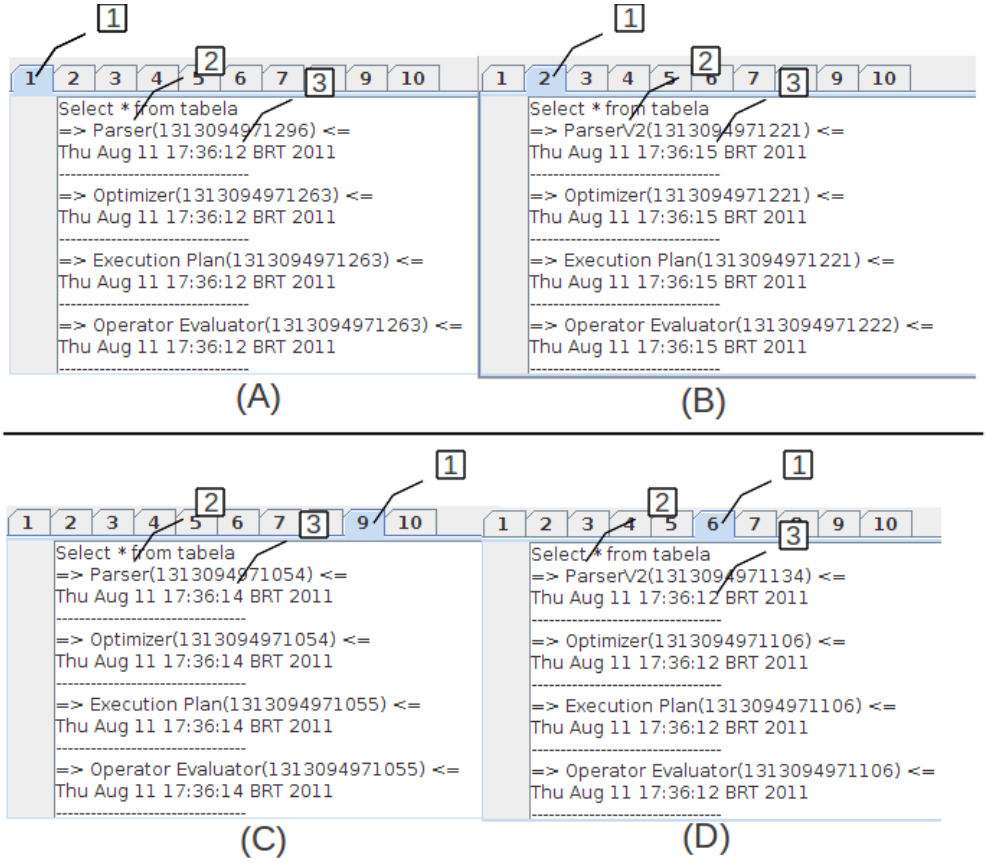

Figura 5.2: Console de resultados produzidos pela execução do protótipo com FRACTAL

Comparando com a Figura 5.2-(B) que se refere a segunda requisição (destaque (1)), verifica-se que o subcomponente que executou foi o *ParserV2*, conforme mostra o destaque (2), em um tempo diferente da Figura 5.2-(A-3). Isto significa que entre a execução as duas requisições ocorreu a substituição dos subcomponentes.

As Figuras 5.2-(C) e 5.2-(D) que se referem às requisições 9 e 6 que executaram com os subcomponentes *Parser* e *ParserV2*, respectivamente, confirmando as trocas durante o processamento de todas as requisições.

5.3.2 Manutenabilidade

A manutenabilidade é uma medida de qualidade de *software* que visa determinar a capacidades deste *software* sofrer correções, modificações ou ampliações (REZENDE, 2006). Brusamolin (2004) aborda métricas de manutenabilidade a nível de código, a nível de projeto e métricas híbridas. Afirma que o projeto da arquitetura é mais impactante do ponto de vista de manutenabilidade que o projeto do algoritmo de baixo nível. Cita ainda duas métricas, ambas baseadas em fluxos ou caminhos de informação dentro do sistema: complexidade ciclomática e

fluxo de informação.

Já Xavier (2001) propõe métrica de manutenabilidade, de acordo com a ISO/IEC 9126-1, para avaliação de padrões arquiteturais orientados a objetos baseado em dois critérios: modificabilidade e testabilidade.

FRACTAL e sua capacidade de reconfiguração simplificam o processo de manutenabilidade, provendo graus elevados de modificabilidade e testabilidade. Isto porque a granularidade tratada no sistema é o componente. O fato do componente respeitar a *interface* predefinida o credencia a ser incorporado sem problemas à arquitetura.

No trecho de código da Figura 5.3.2, observando da linha 15 até 21, neste exemplo, o sistema decide pela utilização do componente *Parser* ou *ParserV2*. Na linha 22 é instanciado como um componente FRACTAL o resultado da decisão anterior. O componente FRACTAL criado então é adicionado como subcomponente (linha 25). Enfim, um procedimento genérico e válido para toda a arquitetura FRACTAL.

```
1  private boolean dsu() throws Exception {
2   if(doDsu()) {
3    if(withDsu) {
4        // stop root component
5        Fractal.getLifeCycleController(this.root).stopFc();
6
7        // unbind
8        Fractal.getBindingController(parser).unbindFc("optimizer");
9        // unbind
10       Fractal.getBindingController(sgbd).unbindFc("parser");
11
12       // remove subcomponent
13       Fractal.getContentController(root).removeFcSubComponent(parser);
14
15       // create new component
16       if(Control.getClassParser() == "pkgSgbdEmComponentesFractal.ParserV2") {
17           Control.setClassParser("pkgSgbdEmComponentesFractal.Parser");
18       }
19       else {
20           Control.setClassParser("pkgSgbdEmComponentesFractal.ParserV2");
21       }
22       parser = cf.newFcInstance(tParser,"primitive", Control.getClassParser());
23
24       // add new component
25       Fractal.getContentController(root).addFcSubComponent(parser);
26
27       // bind
28       Fractal.getBindingController(sgbd).bindFc("parser", parser.getFcInterface("parser"
             ));
29       Fractal.getBindingController(parser).bindFc("optimizer", optimizer.getFcInterface(
             "optimizer"));
30
31       //  start root component
32       Fractal.getLifeCycleController(root).startFc();
33    }
34   }
35   return true;
36  }
```

Figura 5.3: Método que realiza ADS da classe *Control*

6 CONCLUSÕES E TRABALHOS FUTUROS

ADS não é um assunto novo, mas na prática soluções automáticas são restritas. Desde que a demanda de aplicações críticas que exigem disponibilidade 24 X 7, principalmente aquelas que fazem uso de bancos de dados através da Internet, tem aumentado, é interessante que sistemas computacionais sejam providos de técnicas eficientes, de baixo custo, robustas e transparentes para a atualização de *software* sem a parada do serviço. Este trabalho apresentou uma arquitetura para construção de um SGBD, baseada em componentes de *software* capaz de permitir a sua atualização sem a indisponibilização total do sistema e sem usar *hardware* redundante.

Após as avaliações descritas no capítulo 5 demonstrou-se que a sua implementação é viável, respeitanto as restrições impostas pelo próprio modelo de componentes. As propriedades da solução demonstram vantagens importantes no desenvolvimento com este modelo.

Com relação ao desempenho da solução apresentada, infere-se que a degradação média do desempenho foi de aproximadamente **30%** do protótipo no modelo de componentes em relação ao protótipo OO. Isto significa que o *framework* FRACTAL provoca uma sobrecarga significativa no sistema. Mas esta degradação é aceitável, uma vez que não implica na indisponibilidade total do sistema.

Além disso, a solução deste trabalho propõe a atualização do SGBD em módulos, o que permite atualizar apenas parte do SGBD. Isto siginifica que realização de operações pontuais minimizam o tempo de indisponibilidade, uma vez que substituir um módulo custa menos que substituir todo o SGBD.

Propões-se para trabalhos futuros:

- Desenvolver a mesma solução com outros *frameworks* de suporte ao desenvolvimento baseado em componentes de *software*. Esta sugestão tem como objetivo verificar se é possível minimizar a degradação média de desempenho que foi aferida neste trabalho.

- Ampliar a arquitetura proposta para SGBDs com múltiplos nodos (*cluster* ou rede de computadores). Para sistemas multi-nodo, a atualização deve acontecer em cada um dos nodos. O desafio consiste em gerenciar esta atualização, a fim de determinar quando todos os nodos estão atualizados, momento este que se pode afirmar que o sistema está atualizado.

- A observação de como se comportará a solução proposta através de uma implementação

prática. Esta implementação tem o intuito de refinar a solução, idenficar pontos críticos e apontar novas direções de pesquisa. É necessário observar também o grau de alterações que serão necessárias a fim de converter um SGBD desenvolvido no paradigma OO para o de componentes de *software*.

- Implementar propriedades autonômicas para que a atualização de componentes aconteça, preferencialmente, quando o sistema está ocioso.

Isto se propõe porque o tema deste trabalho é amplo e se percebe que as soluções não são definitivas, nem únicas.

REFERÊNCIAS

BASS, L.; CLEMENTS, P.; KAZMAN, R. **Software architecture in practice**. Second Edition.ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003. 21p.

BRUNETON, E.; COUPAYE, T.; LECLERCQ, M.; QUEMA, V.; STEFANI, J. B. "The FRACTAL component model and its support in Java: experiences with auto-adaptive and reconfigurable systems". **Softw. Pract. Exper.**, New York, NY, USA, v.36, n.11-12, p.1257–1284, 2006.

BRUSAMOLIN, V. Manutenibilidade de software. **Revista Digital Online**, [S.l.], v.2, p.10 – 15, Jan 2004. `http://www.revdigonline.com`.

COLLINS-SUSSMAN, B.; FITZPATRICK, B. W.; PILATO, C. M. **Version control with subversion**. Sebastopol, CA, USA: O'Reilly Media, Inc, 2004.

DANTON, G. **Metodologia científica**. Pará de Minas, MG, Brasil: Virtual Books Online MM Editores Ltda, 2002.

ELMASRI, R.; NAVATHE, S. **Sistemas de banco de dados**. 4.ed. São Paulo: Pearson Addison Wesley, 2005.

FABRY, R. S. How to design a system in which modules can be changed on the fly. In: SOFTWARE ENGINEERING, 2., 1976, Los Alamitos, CA, USA. **Proceedings...** IEEE Computer Society Press, 1976. p.470–476. (ICSE '76).

GHARAIBEH, B.; RAJAN, H.; CHANG, J. M. **A quantitative cost/benefit analysis for dynamic updating**. IOWA, USA: IOWA STATE UNIVERSITY, 2009. COMPUTER SCIENCE. (09-27).

HEINEMAN, G. T.; COUNCILL, W. T. **Component-based software engineering**: putting the pieces together. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001.

HICKS, M.; NETTLES, S. M. Dynamic software updating. **ACM Transactions on Programming Languages and Systems (TOPLAS)**, [S.l.], v.27, n.6, p.1049–1096, Nov 2005.

KEPHART, J. O.; CHESS, D. M. The vision of autonomic computing. **Computer**, Los Alamitos, CA, USA, v.36, p.41–50, Jan 2003.

KRUCHTEN, P. Modeling component with the Unified Modeling Language. In: ICSE 1998 INTERNATIONAL WORKSHOP ON COMPONENT-BASED SOFTWARE ENGINEERING, 1998. **Proceedings...** [S.l.: s.n.], 1998. (ICSE '98).

LÉGER, M.; LEDOUX, T.; COUPAYE, T. Reliable dynamic reconfigurations in the Fractal component model. In: ADAPTIVE AND REFLECTIVE MIDDLEWARE: HELD AT THE ACM/IFIP/USENIX INTERNATIONAL MIDDLEWARE CONFERENCE, 6., 2007, New York, NY, USA. **Proceedings...** ACM, 2007. p.3:1–3:6. (ARM '07).

LYU, J.; KIM, Y.; KIM, Y.; LEE, I. A procedure-based dynamic software update. In: INTERNATIONAL CONFERENCE ON DEPENDABLE SYSTEMS AND NETWORKS (FORMERLY: FTCS), 2001., 2001, Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2001. p.271–284. (DSN '01).

MOREIRA, E. H.; PONS, M. E. D. Novas tecnologias na comunicação empresarial, a intranet como ferramenta da comunicação interna. In: CONGRESSO BRASILEIRO DE CIêNCIAS DA COMUNICAçãO, 26., 2003. **Anais...** [S.l.: s.n.], 2003.

MULLINS, C. **Complexity and DBMS release migration**. Mai/2003, `http://www.craigsmullins.com/dbta_020.htm`.

OREIZY, P.; GORLICK, M. M.; TAYLOR, R. N.; HEIMBIGNER, D.; JOHNSON, G.; MEDVIDOVIC, N.; QUILICI, A.; ROSENBLUM, D. S.; WOLF, A. L. An architecture-based approach to self-adaptive software. **IEEE Intelligent Systems**, Piscataway, NJ, USA, v.14, p.54–62, May 1999.

OREIZY, P.; MEDVIDOVIC, N.; TAYLOR, R. N. Architecture-based runtime software evolution. In: SOFTWARE ENGINEERING, 20., 1998, Washington, DC, USA. **Proceedings...** IEEE Computer Society, 1998. p.177–186. (ICSE '98).

OW2, C. **The Fractal Component Model - Specification**. Em: http://fractal.ow2.org.

PRESSMAN, R. S. **Software engineering**: a practitioner's approach. 5.ed. [S.l.]: McGraw-Hill, 2001.

QUATRANI, T. **Visual modeling with Rational Rose 2000 and UML**. [S.l.]: Addison-Wesley-Longman, 1999. 77p. (Addison-Wesley object technology series).

RAJLICH, V.; BENNETT, K. A staged model for the software life cycle. **Computer**, [S.l.], v.33, n.7, p.66 –71, Jul 2000.

RAMAKRISHNAN, R.; GEHRKE, J. **Database management systems**. 3.ed. Singapura: McGraw-Hill Education, 2003.

RESENDE, A. R. M. d. L.; CUNHA, A. M. d.; RESENDE, A. M. P. d. **Um modelo de processo para seleção de componentes de software**. Lavras, MG: UFLA, 2007. (Série didática de engenharia de software).

REZENDE, D. **Engenharia de software e sistemas de informação**. [S.l.]: BRASPORT, 2006. 101-104p.

SALMI, N.; MOREAUX, P.; IOUALALEN, M. Performance evaluation of Fractal component-based systems. **Annals of Telecommunications**, [S.l.], v.64, p.81–100, 2009. 10.1007/s12243-008-0070-1.

SEGAL, M.; FRIEDER, O. On-the-fly program modification: systems for dynamic updating. **Software, IEEE**, [S.l.], v.10, n.2, p.53–65, Mar 1993.

SILBERSCHATZ, A.; KORTH, H. F.; SUDARSHAN, S. **Sistema de banco de dados**. 5.ed. [S.l.]: Elsevier, 2006.

STOYLE, G.; HICKS, M.; BIERMAN, G.; SEWELL, P.; NEAMTIU, I. Mutatis Mutandis: safe and predictable dynamic software updating. **ACM Trans. Program. Lang. Syst.**, New York, NY, USA, v.29, Ago 2007.

WAHLER, M.; RICHTER, S.; ORIOL, M. Dynamic software updates for real-time systems. In: INTERNATIONAL WORKSHOP ON HOT TOPICS IN SOFTWARE UPGRADES, 2., 2009, New York, NY, USA. **Proceedings...** ACM, 2009. p.2:1–2:6. (HotSWUp '09).

WANG, Q.; SHEN, J.; WANG, X.; MEI, H. A component-based approach to online software evolution: research articles. **J. Softw. Maint. Evol.**, New York, NY, USA, v.18, p.181–205, Mai 2006.

XAVIER, J. R. **Criação e instanciação de arquiteturas de software específicas domínio no contexto de uma infra-estrutura de reutilização**. 2001. Dissertação (Mestrado) — Universidade Federal do Rio de Janeiro, Rio de Janeiro, RJ - Brasil.